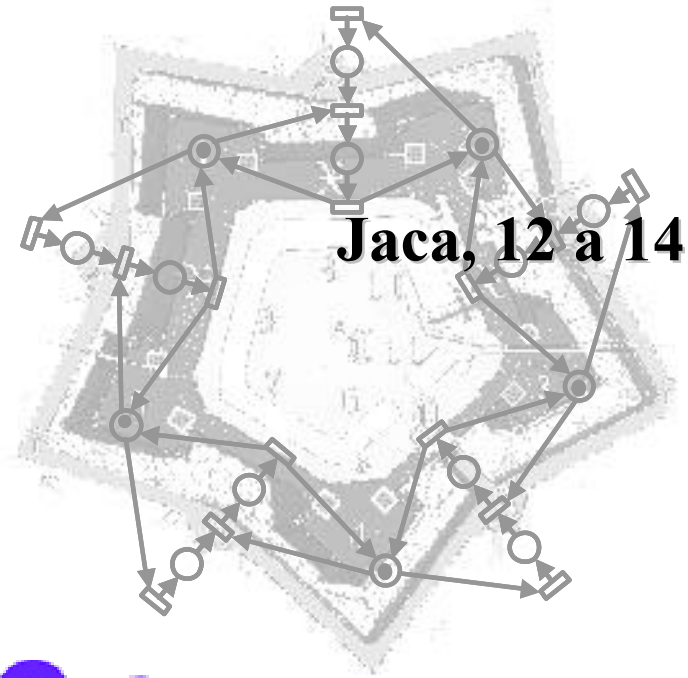
Universidad de Zaragoza

Actas de las X Jornadas de Concurrencia



Jaca, 12 a 14 de junio de 2002

Departamento de Informática e
Ingeniería de Sistemas
Universidad de Zaragoza

Prólogo

En septiembre de 1987 tuvieron lugar en Jaca las Primeras Jornadas de Concurrencia. Este año, 2002, las Jornadas vuelven a Jaca en su décima edición, y el grupo de Métodos Formales de la Universidad de Zaragoza tiene de nuevo el placer de organizarlas.

Desde su inicio, las Jornadas de Concurrencia han tenido dos objetivos fundamentales. El primero, poner en contacto a los grupos españoles cuyos trabajos de investigación estuvieran relacionados con los distintos aspectos de la concurrencia (léase en un sentido amplio que incluye tanto cuestiones relacionadas con el modelado, análisis, implementación y verificación de sistemas concurrentes y distribuidos, como sistemas tiempo real, por ejemplo).

El segundo objetivo de las jornadas era servir como escaparate de los distintos grupos de investigación, de manera que el trabajo de cada uno de ellos pudiera ser conocido por los demás.

Si bien desde el punto de vista técnico se puede apreciar una evolución en los contenidos de las jornadas en los quince años transcurridos (consecuencia de la evolución de las tecnologías), los objetivos siguen siendo los mismos, fundamentalmente debido al éxito alcanzado desde su inicio. Prueba de ello son las cada vez más numerosas actividades de investigación compartidas entre distintos grupos nacionales.

Así pues, estas actas deben verse como un muestrario de lo que actualmente grupos con participantes españoles investigan en temas relacionados con los sistemas concurrentes. Contienen un total de veintisiete ponencias, presentadas por investigadores de catorce universidades españolas y tres centros extranjeros. De entre los trabajos presentados algunos ya han sido dados a conocer en otros foros, nacionales o internacionales, mientras que otros se publican por primera vez. En cualquier caso, es de constatar el amplio conjunto de temas y aproximaciones presentados.

No quisiéramos cerrar este prólogo sin agradecer a todos los participantes el interés mostrado, así como al Ministerio de Ciencia y Tecnología su apoyo económico a través de la acción especial TIC2001-4564-E.

El comité organizador

Javier Campos Laclaustra
Joaquín Ezpeleta Mateo
Jorge Júlvez Bueno
José Merseguer Hernaiz
Carlos José Pérez Jiménez
Fernando Tricas García

Indice

Efficiency and Scalability Models for Heterogeneous Clusters

Luis Pastor and Jose L. Bosque Orero

Universidad Rey Juan Carlos, 28933 Madrid, Spain,
lpastor@escet.urjc.es, jbosque@escet.urjc.es

Abstract. Efficiency and scalability are two key concepts for analyzing the performance of parallel systems. Even though heterogeneous systems are becoming more common, there are no suitable definitions of similar concepts for studying their behaviour. This paper first presents a definition of efficiency that can be applied equally to heterogeneous and homogeneous systems. From that definition, the paper also presents an extension of the isoefficiency method that permits its application to heterogeneous environments as well. Finally, the paper presents some experiments where the method's usefulness and validity are shown.

1 Introduction

Efficiency and scalability are two of the most salient attributes of parallel systems, particularly in the case of clusters. Although more precise definitions will be given later, it is possible to see both as essential qualities of parallel systems (algorithms plus architectures), which manifest how the system behaves when the number of processors is increased. Clusters are naturally very flexible, and the ease with which their size can be changed increases the importance of being able to study the efficiency and scalability of the algorithms to be implemented in these systems [3, 20].

The concept of efficiency is very clearly stated for the case of homogeneous systems [12]. On the other hand, rather than having a definition of scalability which is universally accepted, it is possible to find a number of scalability models that have been proposed during the last years [15–17]. They are typically based on the selection of a metric which is used for characterizing the system's behaviour. The system's performance is then analyzed while the number of processors and the problem size are increased [13]. The system is considered to be *scalable* if the performance measures can be kept constant whenever the number of processors is increased by selecting the appropriate problem size. The system's degree of scalability is given by the ratio *problem growth to system growth* needed to keep the figures constant. Typical metrics that have been proposed include speedup, efficiency, average speed, latency, etc.

All of the metrics and methods for estimating efficiency and scalability which are available nowadays have been developed for the distributed-memory multiprocessor case, where machines are typically composed of a number of identical processors. The situation for clusters is different, since their flexibility permits easy system reconfigurations, which frequently result in heterogeneous systems, integrated by nodes with different features and performance. Consequently, there is a strong need for developing tools and techniques for analyzing the behaviour of heterogeneous environments.

This paper presents a new definition of efficiency which can be applied both to homogeneous and heterogeneous systems, together with a technique for analyzing scalability within heterogeneous systems. Our method, called *heterogeneous isoefficiency*, is based on the isoefficiency technique proposed by Grama, Gupta and Kumar [6].

The rest of the paper is organized as follows: first, a brief summary of scalability models for homogeneous systems is presented, paying particular attention to the *isoefficiency model*. Next, this last paradigm is adapted to the specific nature of heterogeneous systems. Last, some experimental results achieved in the application of this model to a practical problem are presented, together with the main conclusions that can be reached.

2 Background

While efficiency for a parallel homogeneous system is consistently given by the ratio *speedup to number of processors*, a number of techniques have been suggested during the last years for evaluating scalability. For example, in 1994 Sun and Rover proposed the *isospeed* method [23]. Their metric is based on the definition of an *average unit speed* for the system under consideration, which can be taken as the system's achieved speed divided by the number of processors. A parallel system is *scalable* if the average speed can be kept constant while increasing the processor count to N' and increasing the problem size to W'. For them, a *scalability function* $\psi(N, N')$ can be defined as

$$\psi(N, N') = \frac{N'W}{NW'} \tag{1}$$

Sun and Rover's method is simple and easy to use, although its simplicity prevents it from reflecting aspects such as program overhead, system architecture or influence of non-FP operations in the system behaviour.

Zhang, Yan and He suggested also in 1994 an experimental method based on a metric called *latency* [27, 24, 25]. Nussbaum and Agarwal [18] defined the scalability of a particular algorithm and architecture as the ratio of asymptotic speedups for real and ideal machines. Karp and Flatt [14] used the implementations' *serial fraction* as their metric for assessing the operation of a fixed size system, since the existence of increasing values of this parameter indicates poor scalability due to communication overheads when the number of processors is increased

Zorbas *et al* [28] use a *system overhead function* $\Phi(N)$ to account for differences between real and ideal parallel execution times: they define $\Phi(N)$ as the smallest function that can multiply the ideal execution times to yield an upper bound for the real execution times, when N is varied. A parallel system is scalable if the *system overhead function* remains constant when the problem size and number of processors are increased. How much the problem has to be increased to compensate for the inclusion of additional processors determines the system's degree of scalability.

2.1 The Isoefficiency Model for Homogeneous Systems

Grama, Gupta and Kumar proposed in [6] the isoefficiency model for homogeneous parallel systems. Their method, very frequently used for analyzing applications [7–11,

15], constitutes the starting point which will be extended for heterogeneous systems in this paper.

Grama, Gupta and Kumar start with a parallel system, constituted by *N* identical processors linked by a communication network and by a parallel algorithm, constituted in turn by a set of processes which are concurrently executed to solve a unique common problem. Let the *sequential execution time* T_1 be the response time for the best sequential algorithm executed over just one processor, let the *algorithm overhead* T_o be the sum of the times spent by all processors while performing work which is not necessary in the sequential algorithm, and let the *parallel execution time* T_R be the response time corresponding to the execution of the algorithm on a parallel system composed of *N* identical processors.

These terms can be related:

$$N \cdot T_R = T_1 + T_o \tag{2}$$

If we describe the problem size by W, the number of basic operations performed by the best sequential algorithm, then T_1 will be given by the product of W, the problem size, and t_c, the average cost of executing a basic operation.

The isoefficiency model is based on two considerations:

- For a given instance of a particular problem, the speedup does not grow linearly with the number of available processors due to communication overheads and idle times, as given by Amdahl's law [1]. As a result, the efficiency achieved by the system decreases when the number of available processors in a parallel system is increased.
- For a given processor count, larger problem instances result in higher speedups and efficiencies.

Since *problem size* and *processor count* have opposite effects on the efficiency figures, it seems reasonable that a simultaneous increase in both factors might keep efficiency constant. Some algorithm-architecture combinations meet this condition and can be referred to as *parallel scalable systems*. The law that governs how much the problem size W has to increase to cope with additional processors determines how scalable a system is.

Expressing efficiency as:

$$E = \frac{1}{1 + \frac{T_o}{t_c W}} \tag{3}$$

allows us to express W as a function of the algorithm overhead:

$$\frac{T_o}{W} = t_c \frac{1-E}{E} \;\Rightarrow\; W = \frac{T_o}{t_c}\left(\frac{E}{1-E}\right) = K \cdot T_o(N) \tag{4}$$

Where $T_o(N)$ expresses that the algorithm overhead depends on N. Equation (4) determines how much W has to grow in order to keep the efficiency constant whenever *N* is increased and gives consequently the *isoefficiency function* for the system under consideration. It is not possible to obtain isoefficiency functions for non-scalable systems.

3 The Isoefficiency Function for Heterogeneous Clusters

This paper presents a scalability model for heterogeneous clusters based on Grama, Gupta and Kumar's work for homogeneous clusters [6]. In their proposal, the isoefficiency functions depend only on the number of system processors. For heterogeneous systems the situation is different, since processor *nature* affects overall system performance just as much as processor *number*: the fact that nodes with different processor power, I/O capabilities and memory size and speed can be included in the cluster makes it necessary to look for an alternative formulation for the isoefficiency function. Additionally, other aspects such as computational power and efficiency have to be redefined to be applicable to heterogeneous systems.

3.1 Preliminary Definitions

As in [6], let *W* be the problem size, expressed as the number of basic operations needed to solve the problem, and let T_i, the response time for node *i*, be defined as the time passed since the application is launched until the results from node *i* are produced. T_i can be decomposed into computation and overhead times: $T_i = t_c^i + t_o^i$. The response time for an N-processors system, T_R, can be defined as the total time elapsed from the instant when the application is launched until it is completely finished. The response time will depend on the last node to finish:

$$T_R = max_{i=1}^{N} T_i \tag{5}$$

Definition 1 The *average computational power* of node i in a heterogeneous system, denoted by P_i, can be defined as the amount of work finished during a unit time span in that processor.

P_i depends on the node's physical features (CPU organization and speed, memory and I/O capabilities, etc) but also on the particular algorithm implementation that is actually being processed. This is so because different algorithm implementations have different mixtures of basic operations such as memory accesses, fixed and floating point operations, and the performance of different nodes is also dependent on the nature of the basic operation under consideration. The average computational power for *i* under a specific workload W can be expressed as:

$$P_i = \frac{W}{T} \tag{6}$$

Where *W* is taken as the number of elemental operations performed in a specific task on *T* time units. For homogeneous systems, the power is constant: $\forall i, P_i = P$

From a practical point of view, one way to find out the different nodes' relative power is to execute a sequential version of the algorithm under consideration in each of them using a problem size which is large enough to prevent estimation errors due to cache memory effects.

Definition 2 The *total computational power* of a heterogeneous system composed of N processors, denoted by $P_T(N)$, can be defined as the sum of the computational power of all the processors that compose the system.

This parameter can give an idea of the amount of work that a system can perform in a time unit, executing a particular algorithm. This can be written as:

$$P_T(N) = \sum_{i=1}^{N} P_i \tag{7}$$

For a homogeneous system, the total computational power will be equal to $P_T(N) = N \cdot P_i$.

3.2 Definition of Heterogeneous Efficiency

The total computational power of a system depends on the number of processors that compose it as well as on the power of each of them. This means that two systems with the same number of processors do not necessarily have the same total power, which is possible to increment either by increasing the number of processors *(physical scalability)*, or by increasing the power of some of them *(power scalability)* [26].

A major consequence of this fact is that it is not possible to define efficiency for heterogeneous systems as a function of the number of nodes as in the homogeneous case, because the system performance will depend on *which* nodes are actually being used. A definition of efficiency applicable to homogeneous and heterogeneous systems has to take into consideration the extent to which a particular implementation is taking advantage of the system's total computational power for doing useful work rather than for wasting it on communication and synchronization overheads or on processors idling because of load unbalance.

Definition 3 The *efficiency* of a parallel (either homogeneous or heterogeneous) system can be defined as the ratio between the best response time achievable for solving a specific problem in that system and the real response time achieved during the algorithm execution.

We will denote this efficiency as EF in order not to confuse it with the classical efficiency described previously. The best response time will be obtained if the workload is evenly distributed among all of the nodes (perfectly balanced distribution) and if there is no overhead time. This is equivalent to executing the algorithm on a single processor with a computational power equal to the parallel system's total computational power. This can be expressed as:

$$EF = \frac{Optimal\ achievable\ time}{Actual\ response\ time} = = \frac{W}{T_R \cdot \sum_{i=1}^{N} P_i} = \frac{W}{T_R \cdot P_T(N)} \tag{8}$$

For homogeneous systems EF becomes the traditional efficiency,

$$E = \frac{SP}{N} \quad \Rightarrow \quad \frac{T_1}{T_R \cdot N} \tag{9}$$

Where *SP* is the achieved speedup.

$$EF = \frac{W}{T_R \cdot P_T(N)} = \frac{W}{T_R \cdot N \cdot P_i} \quad \Rightarrow \quad \Rightarrow EF = \frac{T_1}{T_R \cdot N} \tag{10}$$

3.3 Heterogeneous Scalability

Definition 4 Given a heterogeneous parallel system $S(N, P_T, W)$ with N processors, a total computational power P_T and a total amount of work represented by W and $S'(N', P'_T, W')$, a scaled system with $P'_T > P_T$, it can be said that S is a *scalable system* if, whenever the system is enlarged from S to S', it is possible to select a problem size W' such that the efficiencies of $S(N, P_T, W)$ and $S(N', P'_T, W')$ are kept constant.

Analyzing the proposed definition of efficiency in equation (8), it is possible to see that it depends on three factors:

- The computational workload. If the work to be performed is increased without changing any of the other parameters, the efficiency increases. Conversely, if the amount of work to be processed decreases, the efficiency decreases as well.
- The system's total computational power, which depends on the number and individual power of the available nodes. This is a parameter that depends both on the system and on the algorithm. Nevertheless, it can be measured or estimated for each particular algorithm implementation.
- The system response time which, as stated in (5), corresponds to the last node to finish. This depends on the system, on the algorithm and on the selected workload distribution strategy.

To minimize the system response time, all of the processors should finish at the same time, which requires a well-balanced workload distribution. Even for applications where the computations' fine grain nature allows us to perform perfectly balanced workload distributions, the distribution process itself introduces overheads that can severely affect the system behaviour.

Depending on the workload distribution and on its associated overheads, a particular combination *algorithm-architecture* will show a different scalability degree. Assuming a fine-grain perfectly divisible workload, our definitions of efficiency and scalability allow us to perform theoretical studies on system scalability. In particular, in this section we will present the results achieved for three different cases: constant overhead, uniformly distributed among processors; overhead constituted by constant and computational-power proportional terms, and finally, overheads constituted by constant and workload proportional terms. Due to space restrictions, only the results will be given. A more detailed analysis and proofs can be found in [19].

1. Given a heterogeneous system $S(N, P_T, W)$, a scaled system $S'(N', P'_T, W')$ and a workload divisible *ad infinitum* and distributed among nodes, $W = (w_1, ..., w_N)$. If the workload has been evenly distributed according to each node's computational power and the overhead times are constant for all of the processors, $t_o^i = C_0 \; for \; i = 1, ..., N$, then both systems have the same efficiency, if and only if:

$$W' = W \frac{P'_T \cdot C'_0(N')}{P_T \cdot C_0(N)} \tag{11}$$

   Equation (11) gives us the heterogeneous isoefficiency function for the specific assumptions of evenly distributed workload and uniformly distributed overheads.

2. For the same conditions as in (1), if the overhead times can be expressed as a sum of a first term, constant for all of the processors, plus a second term, proportional to each node's own computational power, $t_o^i = C_0 + C_1 P_i \; for \; i = 1, ..., N$, then the heterogeneous isoefficiency function is

$$W' = W \cdot \frac{C_0 P'_T + C_1 \sum_{i=1}^{N'} (P'_i)^2}{C_0 P_T + C_1 \sum_{i=1}^{N} (P_i)^2} \tag{12}$$

3. For the same conditions as in (1) and (2), if the overhead times can be expressed as a sum of a first term, constant for all of the processors, plus a second term, proportional to each node's workload, $t_o^i = C_0 + C_1 \frac{W}{P_T} P_i \; for \; i = 1, ..., N$, then the heterogeneous isoefficiency function is

$$W' = W \frac{C_0 P'_T}{C_0 P_T + C_1 \frac{W}{P_T} \sum_{i=1}^{N} P_i^2 - C_1 \frac{W}{P'_T} \sum_{i=1}^{N'} (P'_i)^2} \tag{13}$$

In general, different combinations algorithm-architecture will present overhead distributions that will follow different laws, thereby affecting the system's scalability. The following section shows the experimental results obtained for a real case, together with the predictions carried out with the method proposed in this paper.

4 Experimental Results

A practical experiment was set up in order to test the analytical results presented in the previous section and to show how the heterogeneous isoefficiency model could be applied. For this purpose, a classical application based on a master-slave architecture was selected, and a theoretical scalability analysis was performed. After its implementation, real scalability results were computed for a number of cases in order to validate the theoretical studies.

Within the experiment, two sets of measurements were taken, using homogeneous and heterogeneous clusters respectively, while modifying both the clusters and the problem size.

4.1 System Architecture

The tests were performed over a 25 node PC cluster connected through a 100 Mb/s bus topology Ethernet. Each node is a 266 MHz Pentium II processor with 128 MB main memory, 512 KB cache memory and a 4 GB IDE disk connected through DMA at 16.6 MB/s. The PC's operating system is Linux v. 2.2.12. The application was developed using GNU tools and the MPI/LAM 6.3 library [5, 21, 22].

The application is basically composed of a computing process called *solver*, which has to process a large data set, arranged in the present implementation as a large number of files. The *solver* is replicated in all the computation nodes. A farm strategy was selected to distribute the data set: a *master* process is executed on a central node, which is in charge of the distribution of the work-packages among the different *slave* nodes.

N Nodes	N Files	Exe Time	Speedup	Classical Ef.	Proposed Ef.
2	2	2.04	1.38	0.690	0.692
2	4	3.37	1.67	0.834	0.836
2	8	6.48	1.73	0.865	0.870
2	16	12.43	1.81	0.903	0.907
2	32	24.57	1.84	0.920	0.917
2	64	49.64	1.81	0.907	0.908
2	128	97.33	1.85	0.925	0.926
2	256	196.22	1.83	0.917	0.919
2	512	389.69	1.85	0.924	0.925
2	1024	786.40	1.84	0.919	0.917
4	4	2.34	2.40	0.600	0.602
4	8	3.52	3.18	0.796	0.800
4	16	6.62	3.39	0.848	0.851
4	32	12.73	3.55	0.888	0.885
4	64	24.83	3.63	0.907	0.907
4	128	49.14	3.66	0.916	0.917
4	256	97.67	3.69	0.921	0.923
4	512	195.25	3.69	0.922	0.923
4	1024	394.21	3.67	0.916	0.915
8	8	2.50	4.48	0.560	0.563
8	16	3.96	5.67	0.709	0.712
8	32	7.00	6.46	0.807	0.805
8	64	13.07	6.89	0.861	0.862
8	128	25.46	7.07	0.884	0.885
8	256	49.70	7.24	0.905	0.907
8	512	98.24	7.33	0.916	0.918
8	1024	198.41	7.28	0.910	0.909
16	16	2.89	7.76	0.485	0.487
16	32	5.07	8.92	0.557	0.556
16	64	7.96	11.31	0.707	0.708
16	128	14.50	12.41	0.776	0.777
16	256	27.40	13.14	0.821	0.822
16	512	52.27	13.77	0.861	0.862
16	1024	101.23	14.27	0.892	0.89
16	2048	197.45	14.73	0.921	0.913
16	4096	393.89	14.75	0.922	0.915

Table 1. Comparison of classical and heterogeneous efficiency figures for homogeneous cluster for different problem and cluster sizes.

The master sends the work-packages to the slave processes and then waits to gather the results provided by each slave. Slave processes have to store the received file on the local node, start the execution of its solver instance using previously received data, and return the computed results to the master process. The application allows different solvers to be used, keeping the processing structure independent of the data processing algorithms.

In order to make the cluster heterogeneous, extra load was introduced by executing different programs locally in each node. Consequently, differently loaded nodes yielded different response times since the machine had to share its resources with other processes.

A theoretical analysis of the application communication patterns shows that the overhead time has three components:

- A first term, identical for each of the cluster nodes, corresponding to the node initialization process.

- A second term, directly proportional to the number of nodes composing the system, corresponding to the intercommunication processes between the master node and each of its slaves.
- A third term, directly proportional to the number of data packages that have to be processed by each of the system nodes, which can be expressed as $\frac{W}{P_T} \cdot P_i$, corresponding mainly to the transmission of the data packages.

As a result, we can express the overhead time as:

$$T_o = C_0 + C_1 \cdot N + C_2 \cdot P_i \cdot \frac{W}{P_T} \tag{14}$$

After implementation, the constants C_0, C_1 and C_2 were experimentally estimated, yielding the following figures:

$$C_0 = 0.2 \qquad C_1 = 1.2 \qquad C_2 = 3.2$$

4.2 Isoefficiency Function

For an experimental setup such as the one described above, it is possible to analytically deduce the expression that gives the heterogeneous isoefficiency function. Since it is also possible to perform tests varying both the problem and the cluster size, it is therefore possible to verify experimentally the validity of the proposed models.

From equation (14), given a heterogeneous system $S(N, P_T, W)$, a scaled system $S'(N', P'_T, W')$ and a workload divisible *ad infinitum* and distributed among all of the nodes $W = (w_1, ...w_N)$, if the total workload is evenly distributed among all of the nodes proportionally to their computational power, it can be shown [19] that:

1. The response time is identical for all of the nodes:

$$T_1 = T_2 = ... = T_N = T_R = = \frac{W}{P_T} + C_0 + C_1 \cdot N + C_2 \cdot \frac{W}{P_T^2} \sum_{i=1}^{N} P_i^2 \tag{15}$$

2. The heterogeneous isoefficiency function is

$$W' = W \frac{C_0 P'_T + C_1 N' P'_T}{C_0 P_T + C_1 N P_T + C_2 \frac{W}{P_T} \sum_{i=1}^{N} P_i^2 \; - \; C_2 \frac{W}{P'_T} \sum_{i=1}^{N'} (P'_i)^2} \tag{16}$$

4.3 Experimental Results

A number of tests were performed with the previously described setup by selecting different numbers of nodes and problem sizes, specified in this case by the number of equally-sized data files that the application has to process. As stated before, the tests were performed first in a homogeneous cluster, which was made heterogeneous by including different local loads in each of the cluster nodes.

Three objectives were pursued in these tests:

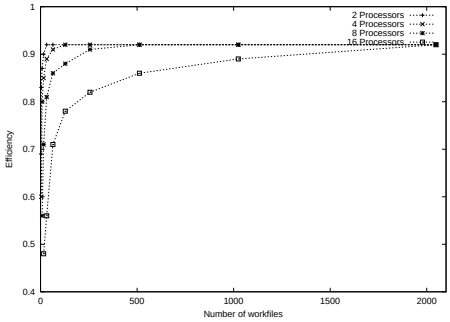


Fig. 1. Classical efficiency in a homogeneous cluster

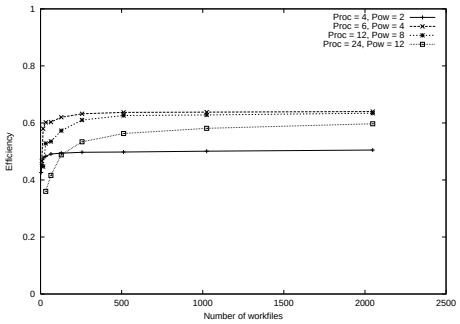


Fig. 2. Classic efficiency in a heterogeneous cluster.

1. Verify experimentally that the heterogeneous efficiency definition introduced in this paper can be applied both to homogeneous and heterogeneous systems, and becomes the classical efficiency whenever the cluster becomes homogeneous. Even though this was analytically shown in section 3.2, it has to be noted that equation (8) needs measures or estimates of each node's computational power. Therefore classical and heterogeneous efficiencies for homogeneous clusters are not guaranteed to coincide.
2. Verify that the homogeneous and heterogeneous isoefficiency functions can be applied to model and predict the systems' scalability (in the case of the heterogeneous isoefficiency function, it can be applied both to homogeneous and heterogeneous systems).
3. Show that it is not possible to apply the classical scalability and isoefficiency definitions to heterogeneous clusters, and that substantial errors are obtained if the proposed heterogeneous modifications are not introduced.

Table 1 presents the execution time, speedup, classical efficiency and heterogeneous efficiency for the homogeneous cluster described above under variable node count and workload conditions. The heterogeneous efficiency figures were computed treating the

homogeneous cluster as if it were heterogeneous, including therefore an experimental stage for the estimation of each node's computational power. It can be seen that the efficiency figures yielded almost identical values.

Figure 1 plots the classical efficiency dependence on node count and workload size, allowing the verification of the homogeneous system scalability. Analyzing Table and Figure 1, it is possible to verify that the isoefficiency function is quadratic $O(N^2)$ with respect to the system's number of nodes as can be analytically deduced. Table 2 also summarizes this result, listing the problem size (number of files) needed to keep efficiency near 0.9 when the number of nodes is increased from 2 to 24.

Table and Figure 3 show similar results for a heterogeneous cluster. It has to be noted that the relevant parameters for analyzing scalability are *number of nodes* as well as *computational power*. It can be seen that the efficiency figures computed using the classical efficiency definition are not consistent (figure 2). The proposed efficiency, nevertheless, yields results which are very similar to those presented for the homogeneous cluster. Additionally, to keep efficiency constant, the problem size has to be increased linearly with respect to the product node of count and total computational power, as predicted analytically in (16).

When analyzing Table 3 it has to be noted that the *Total Power* column gives figures which are not normalized (the rating for a single processor is around 0.72, depending on each node). Also, the column *Estimated N of Files* gives the number of files according to (16) which are necessary to keep the efficiency around 0.9.

N Nodes	N Files	Exe Time	Speedup	Classical Ef.	Proposed Ef.
2	16	12.43	1.81	0.903	0.907
4	64	24.83	3.63	0.907	0.907
8	256	49.70	7.24	0.905	0.907
16	1024	101.23	14.27	0.892	0.89
24	3072	201.3	21.59	0.898	0.896

Table 2. Cluster and workload size needed for keeping efficiency around 0.9 in a homogeneous cluster.

5 Conclusions and Future Work

This paper presents a new definition for efficiency which is applicable both to homogeneous and heterogeneous systems, and fully coincides with the classical efficiency definition for homogeneous systems. The experimental setup shows that whenever a homogeneous system becomes heterogeneous by introducing variable amounts of unbalance among nodes, the heterogeneous efficiency keeps providing results which are consistent with the expectations arisen from executions in homogeneous environments.

It has to be pointed out that the concept of efficiency is central for analyzing the behaviour of parallel systems, and the availability of an efficiency definition applicable to heterogeneous systems allows additional analysis to be performed as in this paper with respect to the scalability of heterogeneous systems.

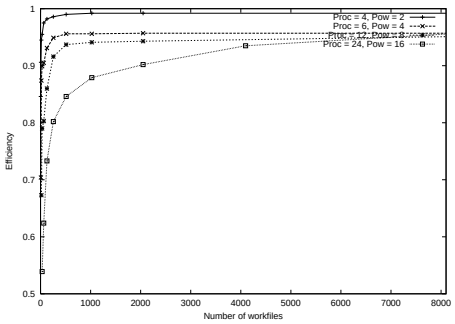


Fig. 3. Proposed efficiency in a heterogeneous cluster.

Comparing the efficiency definition given in this paper with others such as those presented in [25, 4, 2], some remarks can be made:

- The efficiency definition in equation (8) does not need to refer processor performances to any reference processor. This is important, since processor technology is still evolving at a tremendous rate, and any particular reference processor might become obsolete and removed from the cluster in a short time span.
- If we are able to estimate the workload magnitude W for large problem sizes, equation (8) does not need to obtain execution times over any single reference processor. It has to be noted that for large problem sizes obtaining execution times on a single processor might be unfeasible.

This paper also presents a scalability model that can be applied to homogeneous as well as to heterogeneous systems and can be used for predicting algorithm scalability without the actual implementation of the algorithm on the selected architecture. Since it is an *a priori* method, it is more convenient than others that had been previously proposed.

Future work includes the development of more systematic and precise methods for estimating both overhead and relative node computational power.

N Nodes	Total Power	N Files	Estimated N Files	Exe Time	Speedup	Classical Ef.	Proposed Ef.
4	1.44	16	–	12.43	1.81	0.452	0.907
6	2.84	64	65.02	24.89	3.62	0.603	0.905
12	5.68	256	256.58	49.41	7.29	0.607	0.906
24	11.44	1024	1026.48	99.83	14.47	0.603	0.903

Table 3. Cluster and workload size for keeping efficiency around 0.9 in a heterogeneous cluster.

Acknowledgements

This work has been partially funded by the Spanish Commission for Science and Technology (grants CICYT TIC98-0272-C02-01 and TIC99-0947-C02-01). The authors gratefully acknowledge the help provided by Antonio Fernandez Anta debugging and discussing the model and by the reviewers for improving this paper with their suggestions.

References

1. Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Proc. of the SJCC*, 31:483–485, 1967.
2. Jose L. Bosque and L. Pastor. Load balancing on heterogeneous clusters using an heuristic index for machine rating. *European Parallel and Distributed Systems, Euro-PDS'98*, Vienna, Austria, July 1998.
3. Rajkumar Buyya, editor. *High Performance Cluster Computing, Volume 1: Architecture and Systems*. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1999.
4. J. Chen and V. Taylor. Mesh partitioning for distributed systems. *Proceedings of Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.
5. MPI Forum. A message-passing interface standard. 1995. http://www.mpi-forum.org.
6. Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(3):12–21, August 1993.
7. Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. TR 94-63, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, December 1994.
8. Anshul Gupta and Vipin Kumar. On the scalability of FFT on parallel computers. In *Proceedings of Frontiers'90, 3rd Symp. on the Frontiers of Massively Parallel Computation*. NASA/IEEE, October 1990.
9. Anshul Gupta and Vipin Kumar. The scalability of matrix multiplication algorithms on parallel computers. Technical Report TR 91-54, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1991.
10. Anshul Gupta and Vipin Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.
11. Anshul Gupta and Vipin Kumar. Scalability of parallel algorithms for matrix multiplication. In P. Bruce Hariri, Salim; Berra, editor, *Proceedings of the 1993 International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, pages 115–123, Syracuse, NY, August 1993. CRC Press.
12. Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, New York, 1993.
13. P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):589–603, June 2000.
14. Alan H. Karp and Horace P. Platt. Measuring parallel processor performance. *Communications of the ACM*, 22(5):539–543, May 1990.
15. Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. Technical Report 91-55, CS Dept., University of Minnesota, 1991.

16. Vipin Kumar and Anshul Gupta. Analysis of scalability of parallel algorithms and architectures: A survey. In *Proceedings of the 1991 International Conference on Supercomputing (ICS'91)*, Cologne, Germany, June 1991. ACM. Analyzing Scalability of Parallel Algorithms and Architectures TR-91-18 revised November 1992.
17. Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. Technical Report TR-91-18, Computer Science Department, University of Minnesota, June 1991.
18. Daniel Nussbaum and Anant Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, 1991.
19. Luis Pastor and Jose L. Bosque. Efficiency and scalability models for heterogeneous clusters. 2001. Technical Report, Computer Engineering Department, University Rey Juan Carlos, Spain.
20. Gregory F. Pfister. *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing, 2nd ed.* Prentice Hall, Englewood Cliffs, NJ, 1995 edition, 1998.
21. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.
22. Jeffrey M. Squyres, Kinis L. Meyer, M.D. McNally, and Andrew Lumsdaine. Lam/mpi user guide. 1998.
23. Xian-He Sun and Diane T. Rover. Scalability of parallel algorithm-machine combinations. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):599–613, June 1994.
24. Yong Yan, Xiaodong Zhang, and Qian Ma. Software support for multiprocessor latency measurement and evaluation. *IEEE Transactions on Software Engineering*, 23(1):4–16, January 1997.
25. X. Zhang and Y. Yan. Modeling and characterizing parallel computing performance on heterogeneous networks of workstations. In *Symposium on Parallel and Distributed Processing (SPDP '95)*, pages 25–35, Los Alamitos, Ca., USA, October 1995. IEEE Computer Society Press.
26. Xiaodong Zhang, Tong Yan, and Qian Ma. Measuring and analyzing parallel computing scalability. *Proceedings of the 1994 International Computing of Parallel Processing*, 2, 8 October 1994.
27. Xiaodong Zhang, Yong Yan, and Keqiang He. Latency metric: An experimental method for measuring and evaluating parallel program and architecture scalability. Technical Report 3, High Performance Computing and Software Laboratory, University of Texas at San Antonio, Texas USA, February 1994.
28. J. R. Zorbas, D. J. Reble, and R. E. VanKooten. Measuring the scalability of parallel computer systems. In ACM, editor, *Proceedings, Supercomputing '89: November 13–17, 1989, Reno, Nevada*, pages 832–841, New York, NY 10036, USA, 1989. ACM Press.

Register Pressure-Based Modulo Scheduling for Clustered VLIW Architectures

Alex Aletà[1], Josep M. Codina[1], Jesús Sánchez[1], Antonio González[1], and David Kaeli[2]

[1] Dept. of Computer Architecture, UPC, Barcelona, SPAIN
aaleta,jmcodina,fran,antonio@ac.upc.es
[2] Dept. of Electrical and Computer Eng., Computer Architecture Research Lab.
Northeastern University, Boston, MA, USA
kaeli@ece.neu.edu

Abstract. This paper presents a new modulo scheduling algorithm that carefully considers register pressure. Modulo scheduling is a commonly used technique for software pipelining multiple iterations of a loop. Our work targets improving instruction schedules on clustered architectures; clustering divides processor resources into groups. This paper describes an improved Data Dependence Graph (DDG) partitioning algorithm and presents a set of effective algorithms for scheduling spill code associated with excess register pressure. We expand the traditional definition of register pressure to include the impact scheduling can have on intercluster register bus traffic. We describe how we estimate pressure during graph refinement and present a greedy spill code scheduling algorithm. We evaluate their benefits when running SPECfp applications on a clustered VLIW architecture. Finally, we evaluate the benefits of code duplication to relieve intercluster register bus pressure.

1 Introduction

We are presently seeing rapid growth in the embedded and low-power processor domains. A number of recent processors exploit a microarchitecture called *clustering*, that physically partitions functional elements and resources. The components of each cluster are simpler and thus consume less power than more unified designs. Cluster components can be laid out close together, which can reduce signal transmission delays [16]. Long wires are used to interconnect clusters. The use of clustering is especially noticeable in the DSP market, including Analog Devices' TigerSHARC [15], BOPS's ManArray[30], HP/ST's Lx [11], and the Equator MAP1000 [26]. All of these processors implement a VLIW architecture, and rely on the compiler to perform instruction scheduling.

The compiler plays a critical role in the success of a clustered VLIW. The compiler must aggressively schedule code to make best use out of the multiple resources provided. In this paper we focus on instruction scheduling for clustered processors. We limit our focus to scheduling software pipelined loops [23], since a majority of the execution on this class of processor is found in loop bodies.

In this work we explore the issues of register pressure and register bus pressure for modulo-scheduled loops. We attempt to maintain a global view of the

program in order to avoid making poor decisions early during scheduling. We describe our algorithms and evaluate them across 678 different loops taken from the SPEC95fp benchmark suite. The first algorithm uses *register pressure estimation* during partitioning decisions. This algorithm is applied during each partition refinement iteration. The second algorithm uses a greedy-based register pressure analysis to schedule spills.

We also investigate the benefits of operation duplication on reducing intercluster register bus communications. These buses are effectively ports on the register file, and present an additional constraint to consider during scheduling. Reducing intercluster communication is a critical issue in clustered architectures and can dramatically affect the length of the final schedule. Using our register-spill algorithms, combined with code duplication, we improve the instruction throughput on a 4-cluster VLIW architecture by as much as 20%.

The remainder of this paper is organized as follows. Section 2 provides an overview of clustered VLIW and modulo scheduling. Section 3 describes the proposed approach. Section 4 reports on the potential benefits of our new algorithms. Section 5 discusses a number of related ideas to those presented. Section 6 summarizes the work and describes directions for further improvement.

2 Overview of the Architecture

Centralized resources tend to increase design complexity and limit the scalability of a design. Clustered VLIW architectures decentralize components. A single cluster is composed of multiple functional units sharing a common register file. We consider three types of functional units: 1) integer arithmetic, 2) floating-point arithmetic and 3) memory access. Multiple clusters share a common memory hierarchy and communicate operands among clusters using a set of dedicated *register buses*. For sake of simplicity, all of the clusters in our design are identical. Figure 1 shows the microarchitecture for the clustered VLIW considered in this work. VLIW instructions are issued to a cluster in a lockstep fashion (all clusters work on the same VLIW instruction together). In each cycle, every cluster will fetch the operations contained in their corresponding part of a VLIW instruction. Additional details of this cluster VLIW processor can be found in [34, 6, 1].

2.1 Modulo Scheduling Background

Modulo scheduling is an instruction scheduling technique for program loops [23]. It has been shown to be a very effective technique for exploiting the available parallelism in cyclic codes. Modulo scheduling attempts to reduce the *Initiation Interval* (II) associated with a loop; the II is a measure of the number of cycles between successive iterations of loop, while respecting data dependencies and resource requirements. For loops with high trip counts, the II can be used directly to calculate the overall runtime of the loop.

High register bus pressure, high register pressure (i.e., many operands live concurrently) and the addition of register spill code can dramatically increase the II [25]. In this work we look to provide an aggressive schedule using graph-based analysis that uses multiple algorithms to better manage these problems.

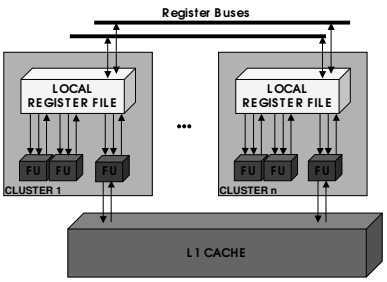


Fig. 1. Clustered VLIW microarchitecture. Register values are communicated through intercluster register buses.

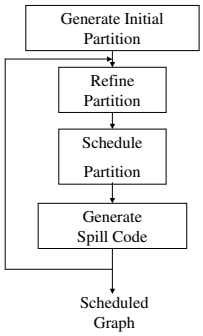


Fig. 2. The high-level structure of our scheduling framework.

Modulo scheduling uses a Data Dependence Graph (DDG) to represent the relationships between different operations in a loop. The set of nodes (V) represents the set of instructions and the set of edges (E) represents the dependencies among these instructions. In a clustered VLIW architecture, we need to partition this graph to map subgraphs on individual clusters.

3 Proposed Algorithm

Figure 2 provides the high-level flow of our proposed algorithm. We perform register pressure estimation during the generation of the initial partition, as well as during each pass through the refinement stage. Once we have scheduled the operations, we utilize greedy spill code generation. When refining the partition we will attempt to perform code duplication to reduce intercluster register bus pressure.

3.1 Generate Initial Partition

Our scheduling algorithm first tries to partition the DDG. Thus, we have a global vision of the graph and we can deal better with the problems related with the split register file. For this purpose we use multilevel partitioning strategies which have been shown to be very effective. They work as follows: first, a new smaller graph (a graph with fewer nodes) is built with a similar structure to the original one. Then, this new graph is partitioned inducing a partition in the original graph. Finally, some heuristics are applied to enhance the partition of the original graph. In practice, the smaller graph is obtained by *coarsening* the original graph, consolidating multiple nodes in a single node, thus maintaining a similar structure as the original graph. Coarsening is repeated iteratively until a graph containing as many nodes as the number of intended partitions (i.e., the number of clusters) is obtained. Then, arrive at an initial partition (i.e., each node is assigned to a different *cluster set* (CS)). The second step in a multilevel partitioning strategy simply induces back the partition of the smaller graphs into the bigger ones and tries to improve the initial partition using heuristics to move nodes among clusters.

Graph Coarsening The coarsening process involves performing a matching in the current graph. For a graph $G = (V, E)$, a matching is a set M of the edges in E such that no pair of edges in M is incident on the same node. Each edge of the graph is weighted using two values: 1) the impact on the schedule if we increase the delay on this edge, and 2) the *slack* time, which represents the number of cycles that could be added to this edge without affecting execution tine. At each step of the coarsening we select the maximum weight matching, defined as the set of nonadjacent edges M such that the sum of their weights is larger than any other match. The nodes incident on these edges will then be fused. We utilize the LEDA library developed in [27] to obtain the matching. After the graph is coarsened such that it contains the same number of nodes as the number of clusters, we attempt to refine the obtained partition.

3.2 Refine Partition

A number of refining algorithms have been proposed for improving a partition, most of them based on the work of Kernighan and Lin [21] and the improvements by Fiduccia and Mattheyes [14] . The general idea in these algorithms is to move nodes from one subgraph to another until some objective is achieved. We apply two different refinements during this process: the first one tries to avoid excessive load in clusters and the second one tries to improve the partition, that is, reduce the execution time of the scheduled loop. In Figure 3 we provide pseudocode for movements during this step.

1. **Best_Movement** - According to the initiation interval (II) and the resources available in the architecture, there is limited space to schedule instructions in each cluster. If the usage of a resource (registers, memory or functional units (FUs)) in a cluster is too high then we try to move nodes that use this resource to other clusters where the load of this resource is lower. Note, we only consider memory and FUs during the refinement of the initial partition since once the partition is balanced[1] movements overloading a cluster are not allowed.
2. **Best_Adjacent_Movement** - At every step we try to enhance the partition. For this purpose we move nodes to adjacent clusters, then compare the different partitions obtained and finally select the best overall movement. To compare different partitions an aproximate schedule is worked out: first, we compute the new II taking into account bus pressure and delayed recurrences (recurrences with edges in the cut). Afterwards, we try to find a suitable slot for each node as close as possible to his predecessors/successors (not enlarging the lifetime). However, if there is not space for scheduling a node, instead of increasing the II and re-scheduling, we schedule it anyway but further from its predecessors/successors. In case the node we can not schedule belongs to a recurrence we will use a higher II to compare the partition obtained. Thus, estimations on II, length of the scheduling and number of communications are obtained. Furthermore, register pressure can

[1] Balanced means that no cluster is overload.

also be estimated with the aproximate scheduling (with the algorithm in Figure 4). The best partition is the one that minimizes execution time or, in case of a tie, the one that minimizes the split register bank usage. If moving a node from one cluster to another overloads the second cluster we look for a node in the second cluster such that moving it to the first one it balances the partition again.

These two steps have different objectives. Best_Movement attempts to achieve better balance in the partition, whereas Best_Adjacent_Movement looks to refine a partition without undoing the balance achieved by Best_Movement. Thus, Best_Movement will precede Best_Adjacent_Movement. Both refinements are applied sequentially during every partition refinement step.

In previous work [1] we did not consider register pressure during partitioning. Instead, we relied on the *URACAM* modulo scheduler framework [6] to schedule register spills. This had a very negative effect on register pressure since partitioning tended to concentrate a lot of nodes in a single cluster in order to reduce the number of communications. We will use the results developed in [1] as our base configuration, since they represent a good comparison point for modulo scheduled loops on a clustered VLIW processor. We will compare the impact of considering register pressure during each partitioning and refinement pass.

Code Replication Once we generate an initial partition, we also consider if we can reduce intercluster communications by duplicating operations in other clusters. This has the effect of reducing intercluster communications since the data value produced by a duplicated operation will now be available locally. But we need to be careful not to produce pressure on other resources (e.g., registers) for the sake of reducing intercluster communications. Thus, we do not attempt to replicate code if there are already sufficient register communication slots available.

Code replication works as follows. Each node in the graph that *produces* a value that needs to be *consumed* by another cluster is a candidate for replication. We first test if replication is possible, and if it is possible, we compute the reduction in communication due to the replicated code. To replicate a node, we first have to test for two criteria:

1. a functional unit slot is available in the consumer cluster to execute the operation, and
2. registers are available in the consumer cluster to hold the variable.

A node will be replicated in those clusters where a consumer is scheduled. We utilize a greedy heuristic, finding the locally optimal node to replicate. Among all replications possible, we choose the replication that introduces the least additional pressure on resources. Replication continues until we have tried all possible copies and no more can be made.

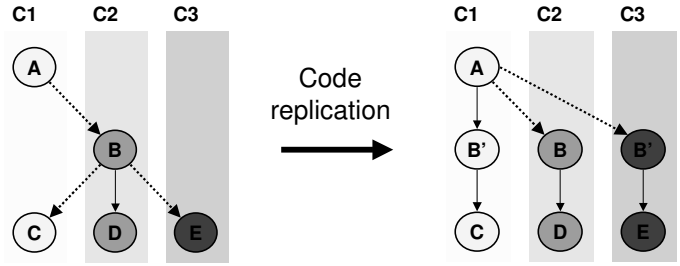
Figure 5 shows an example and pseudocode for our code replication algorithm. In this example, there are two nodes A and B that communicate a value. A produces a value that is consumed by B and a new value is produced by B.

```
BEGIN Best_Movement:
 While (Pressure too high on FUs/memory and
        not every move attempted without improvement)
    {Foreach cluster
       { Foreach node
            { Try to move any node vi from cluster Cj to any other cluster;
              Compute the resulting FU pressure;
            }
       }
     Pick the best movement;
     Update the partition;
    }

 While (Pressure too high on registers and
        not every move attempted without improvement)
    {Foreach cluster
       { Foreach node
           { Try to move any node vi from cluster Cj to any other cluster;
             Compute the resulting register pressure;
           }
       }
     Pick the best movement;
     Update the partition;
    }
END Best_Movement


BEGIN Best_Adjacent_Movement:
 While (Pressure too high on FUs and
        not every move attempted without improvement)
    {Foreach cluster
        { Foreach node
            { Try to move any node vi from cluster Cj to all other clusters;
              Compute the resulting FU pressure;
            }
        }
     Pick the best movement;
     Update the partition;
    }

 While (Improvement is found)
    {Forall nodes v adjacent to the cut of the partition
       { Try to move vi from cluster Ci to the adjacent cluster Cj;
         If not beneficial
            { Identify adjacent node u in Cj to move to Ci;
            }
       }
     Pick the best movement;
     Update the partition;
    }
END Best_Adjacent_Movement
```

Fig. 3. Pseudocode for our two refinement steps.

```
Foreach clusters
  {sum = 0;
   Foreach node
     { Find the longest lifetime for this node
        (i.e., the longest edge, measured in cycles);
       sum = sum + longest lifetime;
     }
   IIreg(cluster) = sum / # of regs per cluster;
  }
Select the largest IIreg(cluster).
```

Fig. 4. Pseudocode for our register pressure estimation.




```
while (there is improvement) {
  foreach (node that communicates a value) {
    duplicate node in each cluster where there is a consumer;
    compute new number of communications;
    while (improvement is zero){
      duplicate parent node;
      choose only the parent that produces the largest benefit;
    }
 }
  compute the best improvement among all possible duplications;
}
```

Fig. 5. Code replication example and pseudocode.

Node B then stores the value in its local register file for node D to consume, and communicates this value to nodes C and E on clusters 1 and 3, respectively. Assume there is no room on cluster 2 to schedule node A. When we consider duplicating node B, we find we can duplicate this node in both clusters 1 and 3 (where the produced value is consumed). Since there are available resources for both copies, we can consider this replication. In this example we reduce intercluster communications by one access. In the original partition, nodes A and B broadcast their results, but in the new partition, only node A broadcasts its result.

Before entering any refinement steps (except the refinement of the initial partition), we undo any code duplication. We do this to insure that we have not entered a local minimum in the partition space.

3.3 Schedule Partition and Generate Spill Code

Figure 6 shows a more detailed flow of our scheduler. We have just described how we arrive at an initial partition, as well as how we replicate operations. We then check if the initiation interval for this partition (II_{part}) is greater than the minimum initiation interval (MII) computed for a unified VLIW [31]. If $II_{part} > MII$, then we refine the partition until we produce an II that is equal to the II_{part}. Once we arrive at this partition, we then can schedule the code.

The scheduler used in this algorithm has little freedom to make decisions. It is guided by the information obtained from the partitioning. Thus, all communications are fixed and the extra instructions necessary to perform these communications are added to the DDG. Then, we schedule the nodes following the strategy used in the Swing Modulo Scheduling [24], that is by giving priority to the nodes that belong to recurrences. When scheduling the nodes two problems could appear:

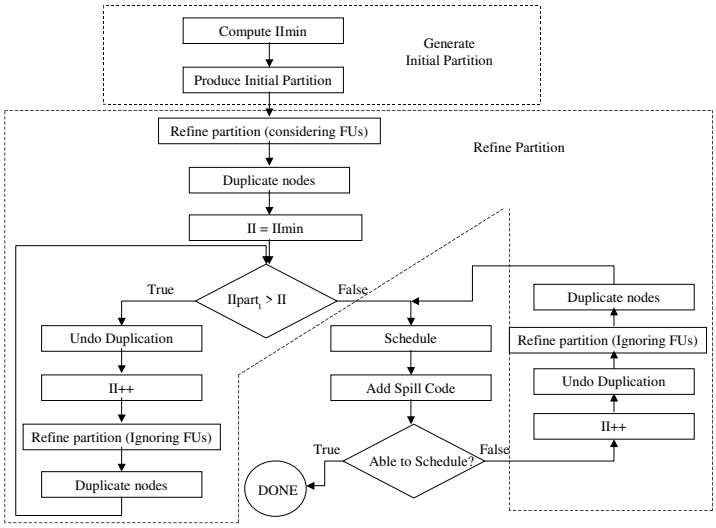


Fig. 6. Detailed description of our complete algorithm.

1. A node in a recurrence can not be scheduled because there are not enough available resources where needed. Then, the initiation interval must be increased.
2. Register pressure is too high. Then, we add spill code. If register pressure is still too high we must increase the initiation interval.

Scheduling Register Spills In our register spill algorithm, we apply a greedy heuristic that selects the potentially most beneficial spill. Our objective function selects the operand to spill that covers the maximum number of cycles where register pressure is a problem. We apply this heuristic, one spill at a time. This is depicted in the righthand side of Figure 6, and we will loop through these steps for each spill until we are able to schedule the code.

Increasing the Initiation Interval When no valid schedule can be obtained the initiation interval must be increased. Therefore, extra resources will appear in each cluster and the old partition may be enhanced, for instance, by reducing the number of communications. Thus, all replicated operations are removed from the graph in order not to over-constrain the problem. Then, the partition is refined again (note that now just the Best_Adjacent_Movement heuristic will be used since there are enough resources in each cluster to schedule the instructions). Afterwards, the graph will be re-scheduled following all the steps described above.

4 Evaluation

Our algorithms have been implemented using the ICTINEO compiler framework [2]. We evaluate 678 loops present in the SPECfp95 benchmark suite. We

Architecture	Clusters	Regs	Register Bus Lat
Arch I	2	64	1
Arch II	4	64	1
Arch III	4	64	2
Arch IV	2	32	1
Arch V	4	32	1

Table 1. Configurations considered in this work.

Resource	Unified	2-cluster	4-cluster
INT/cluster	4	2	1
FP/cluster	4	2	1
MEM/cluster	4	2	1

Table 2. Clustered VLIW resource configurations.

Latency	INT	FP
MEM	2	2
ARITH	1	3
MUL/ABS	2	6
DIV/SQR/TRG	6	18

Table 3. Operation latencies.

study five different architectures, which are defined in Table 1. All architectures assume a single intercluster register bus. Table 2 lists the functional unit resources provided in each cluster. Table 3 provides functional unit latencies. Since we are interested in register spills and intercluster communication, we report on machine configurations which vary parameters related to these design features. We assume a perfect memory system in this work, though plan to consider memory effects in future work (since this impacts the cost of a spill).

For each of these configurations, we present results for our three different scheduling algorithms:

- Baseline - Results reported in [1], which utilized the same compiler infrastructure, but utilize the URACAM [6] scheduler to produce the final schedule without considering register pressure when partitioning the graph.
- Spill - Application of the new algorithm without duplicating nodes, and
- Dupe - Duplication of nodes in other clusters to reduce pressure on the intercluster bus. This result includes our two spill code analyzes.

We also include results for a unified cluster comprised of four functional units of each type and a unified register file.

Our performance metric is the instructions per cycle (IPC). Note that this metric does not consider the impact of the cycle time, which is one of the important benefits of clustering. Thus, the IPC of an equivalent non-clustered architecture can be considered as a lower-bound for the clustered organizations since no communication is needed.

In Figures 7, 8 and 9 we present IPC numbers for the three configurations with 64 registers (ArchI, ArchII and ArchIII respectively). We can see that for all programs except swim (in ArchI with spills) we obtain better results than the baseline. The ArchI (2 clusters, 64 registers, 1 register bus with 1 cycle latency) represents the least restrictive configuration considered (with respect to an equivalent unified architecture with the same number of resources and registers, but without paying any communication overhead). As we can see from Figure 7, the algorithm proposed in this paper (with spill and dupe) obtains IPC results very close to the unified architecture. In fact, the degradation in IPC is

3% on average. From more restrictive configurations (ArchII and ArchIII) the degradation, as expected, increases, but on average we obtain an improvement of about 8%-20% when compared with the baseline algorithm. From these graphs we can also see that, except for particularly cases (swim and turb3d in ArchI, and swim in ArchII), the code duplication optimization improves (or does not degrade) the proposed algorithm. On average, for the three configurations a 2-3% improvement is obtained. Note, however, that for some programs (e.g., turb3d in ArchI, tomcatv, turb3d and fppp in ArchII, and tomcatv and swim in ArchIII) this improvement in more noticeable.

In Figures 10 and 11 we show the results for the two configurations with 32 registers (ArchIV and ArchV). As we saw in configurations with 64 registers, we generally outperform the baseline algorithm, although for some programs (swim and apsi in ArchIV and apsi and fpppp in ArchV) we get particular poorer results.

5 Related Work

Finding an optimal schedule in a resource constrained environment is well known to be NP-complete. For this reason, many heuristics have been proposed in order to find near-optimal schedules. These heuristics have different goals: increasing throughput [18, 31], minimizing register pressure [9, 8], reducing the effect of the cache misses, or improving several of them simultaneously [17, 8, 32, 25]. All of these studies focus on modulo scheduling algorithms targeting unified (i.e., non-partitioned) architectures.

There are several works related to acyclic code scheduling for clustered architectures [10, 4, 7, 19, 29]. The most closely related work to our ideas include the work of Kailas, Ebcioglu and Agrawala [20] where they proposed an approach to cluster assignment, instruction scheduling and register allocation on a single phase, based on a list scheduling scheme. These works differ from the approach presented in this paper in that they target instruction scheduling for acyclic code. Besides, they also use different cluster assignment heuristics. Some modulo scheduling approaches, targeting clustered VLIW architectures, have been recently proposed:

- Nystrom and Eichenberger [28] investigated cluster assignment for modulo scheduling, mainly focusing on minimizing execution overhead due to intercluster communication with a two-step approach: first partitioning the dependence graph of the loop body (assigning each operation to a cluster), and then scheduling the operations following the graph partition.
- Fernandes et al. [13] proposed a modulo scheduling approach integrating scheduling and cluster assignment in a single step. However, they assume an architecture with an unusual register file organization based on a set of local queues for each cluster and a queue file for each communication channel.
- Sánchez and González [34] proposed a unified assign-and-schedule approach in which cluster selection and scheduling are done in a single phase. That work was later extended to deal with a distributed cache memory [33].

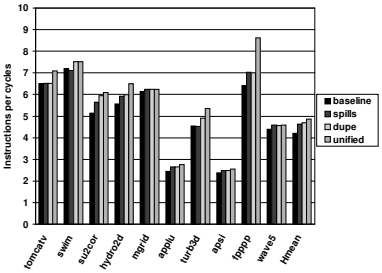


Fig. 7. IPC numbers for an architecture with 2 clusters, 64 registers, 1 register bus with a 1 cycle register bus latency (Arch I)

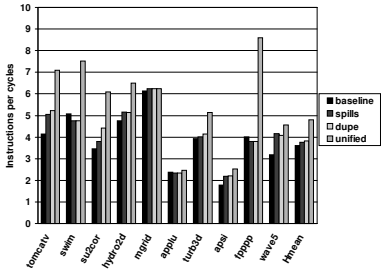


Fig. 10. IPC numbers for an architecture with 2 clusters, 32 registers, 1 register bus with a 1 cycle register bus latency (Arch IV)

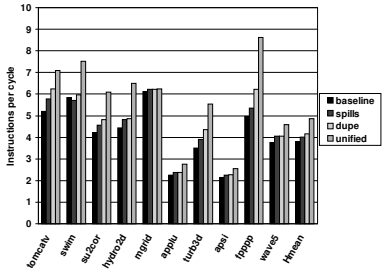


Fig. 8. IPC numbers for an architecture with 4 clusters, 64 registers, 1 register bus with a 1 cycle register bus latency (Arch II)

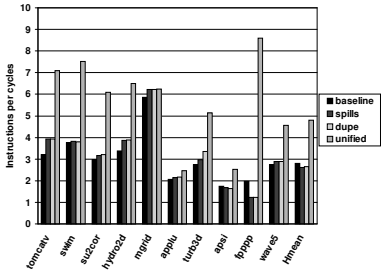


Fig. 11. IPC numbers for an architecture with 4 clusters, 32 registers, 1 register bus with a 1 cycle register bus latency (Arch V)

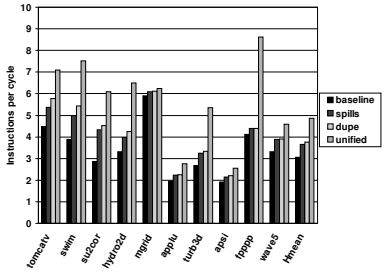


Fig. 9. IPC numbers for an architecture with 4 clusters, 64 registers, 1 register bus with a 2 cycle register bus latency (Arch III)

- Codina et al. [6] presented an approach to deal with instruction scheduling, cluster assignment and register allocation on a single phase with a sophisticated approach to insert spill code on-the-fly and effective mechanisms to deal with communications, register and memory pressure at the same time.
- Zalamea et al. [36] also proposed a technique to cluster assignment, instruction scheduling and register allocation based on an iterative scheme [31] with some heuristics to deal with spill code [35].
- Aletà et al. [1] presented a graph-partitioning based approach with closed interaction to the scheduling phase. The main goal was to improve the results obtained for a technique that combines cluster assignment, instruction scheduling and register allocation in single phase [6] with the global view of the whole problem given by a technique based on a partitioning of graph. This work differs from the approach presented in this paper in that it did not take into account the register pressure while it computes a partition of the graph.

There are a few works related to code replication:

- Chaitin et al. [5], in the context of register allocation based on graph-coloring, point out that some values can be cheaply recomputed instead of spilled to memory. Based on this observation, they proposed a technique called *rematerialization*. This technique was later extended by Briggs et al. [3].
- Kuras et al. [22] describe a technique called *value cloning* for Long Instruction Word architectures with partitioned register banks. They showed how cloning read-only values and induction variables could improve schedules for machines with partitioned resources (registers). They noted that value cloning can be particularly beneficial to resource constrained loops.
- Farkas et al. [12] described the Multicluster, a superscalar architecture with the register file partitioned in two banks. In this design, the communication mechanism is based on instruction copies, distributing a copy to each cluster involved in the transfer.

6 Summary

In this paper we present a new modulo scheduling algorithm that considers register pressure. We also investigated combining operation duplication to reduce register bus pressure. We found that we could significantly improve performance over previously published results.

In this work we performed spill code generation after we had performed graph partitioning and instruction scheduling. While we attempted to reduce spill code generation using a greedy heuristic, it is well known that if spill code generation is analyzed during instruction scheduling, then improved schedules can be produced. This is one focus of our future work.

Acknowledgments

This work has been partially supported by the ESPRIT project MHAOTEU (EP 24942), the Ministry of Science and Technology of Spain and the European

Union (FEDER funds) under contract TIC2001-0995-C02-01, Direcció General de Recerca of the Generalitat de Catalunya under grant 2001FI 00664 UPC APTIND and Analog Devices. David Kaeli is supported by the *Ministry of Education, Culture and Sports* of Spain and the National Science Foundation.

References

1. A. Aletà, J. M. Codina, J. Sánchez, and A. Gónzalez. Graph-Partitioning Based Instruction Scheduling for Clustered Processors. In *Proc. of 34th Int. Symp. on Microarchitecture*, Dec 2001.
2. E. Ayguadé, C. Barrado, A. González, J. Labarta, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera, and M. Valero. Ictineo: A Tool for Research on ILP. In *Supercomputing 96*, 1996.
3. P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proc. of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.
4. A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register File for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proc. of the 25th Int. Symposium on Microarchitecture*, pages 292–300, 1992.
5. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation Via Coloring. In *Computer Languages*, pages 47–57, January 1981.
6. J. M. Codina, J. Sánchez, and A. Gónzalez. A Unified Modulo Scheduling and Register Allocation Technique for Cluster Processors. In *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 175–184, Sept 2001.
7. G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers. Technical Report HP-98-13, HP Labs Technical Report, Jan 1998.
8. A. Eichenberger and E. Davidson. Stage Scheduling: A Technique to Reduce the Register Requirements of a Module Schedule. In *Proc. of the 28th Int. Symposium on Microarchitecure*, pages 338–349, 1995.
9. A. Eichenberger, E. Davidson, and S. Abraham. Optimum Module Schedules for Minimum Register Requirements. In *Proc. of Supercomputing '95*, 1995.
10. J. Ellis. *Bulldog: A Compiler for VLIW Architecture*. MIT Press, Cambridge, MA, 1986.
11. P. Faraboschi, G. Brown, J. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proc. of the 27th Int. Symp. on Computer Architecture*, pages 203–213, June 2000.
12. K. Farkas. *Memory-System Design Considerations for Dynamically Scheduled Microprocessors*. PhD thesis, University of Toronto, 1997.
13. M. Fernandes, J. Llosa, and N. Topham. Partitioned Schedules for Clustered VLIW Architectures. In *Proc., 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'1998)*, pages 386–391, March 1998.
14. C. Fiduccia and R. Mattheyes. A Linear-Time Heuristic for Improving Network Partitions. In *Proc. of 19th Design Automation Conference*, pages 175–181, 1982.
15. J. Fridman and Z. Greenfield. The TigerSharc DSP Architecture. *IEEE Micro*, pages 66–76, Jan-Feb 2000.
16. R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proc. of the IEEE*, pages 490–504, April 2001.
17. R. Huff. Lifetime-Sensitive Modulo Scheduling. In *Proc. of the Int. Conf. on Programming Languages, Design and Implementation*, pages 318–328, 1993.

18. S. Jain. Circular Scheduling: A New Technique to Perform Software Pipelining. In *Proc. of the Int. Conf. on Programming Languages, Design and Implementation*, pages 219–228, 1991.
19. S. Jang, S. Carr, P. Sweany, and D. Kuras. A Code Generation Framework for VLIW Architectures. In *Proc. of the 3rd Int. Conf. on Massively Parallel Computing Systems*, April 1998.
20. K. Kailas, K. Ebcioglu, and A. Agrawala. CARS: A New Code Generation Framework for Clusterd ILP Processors. In *Proc. of the 7th Int. Symposium on High Performance Computer Architecture*, pages 133–143, 2001.
21. B. Kernighan and S. Lin. An Effective Heuristic Procedure for Partitioning Graphs. *Bell Syst. Tech. Journal*, pages 291–307, 1970.
22. D. Kuras, S. Carr, and P. Sweany. Value Cloning For Architectures with Partitioned Register Banks. In *Workshop on Compiler and Architecture Support for Embedded Systems*, pages 1–5, Dec 1998.
23. M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. of the 8th Int. Conf. on Programming Languages, Design and Implementation*, pages 258–267, June 1988.
24. J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing Modulo Scheduling: A Lifetime Sensitive Approach. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 1996.
25. J. Llosa, M. Valero, E. Ayguadé, and A. González. Modulo Scheduling with Reduced Register Pressure. *IEEE Transactions on Computers*, 47(6):625–638, 1998.
26. MAP1000. MAP1000 unfolds at Equator. *Microprocessor Report*, 12(16), Dec 1998.
27. K. Mehlhorn and S. Naher. LEDA: A Library of Efficient Data Structures and Algorithms. *ACM Communications*, 38:96–102, 1995.
28. E. Nystrom and A. E. Eichenberger. Effective Cluster Assignment for Modulo Scheduling. In *Proc. of the 31st Int. Symposium on Microarchitecture*, 1998.
29. E. Ozer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *Proc. of the 31st Int. Symposium on Microarchitecture*, pages 308–315, 1998.
30. G. Pechanek and S. Vassiliadis. The ManArray Embedded Processor Architecture. In *Proc. of 26th Euromicro Conference*, pages 348–355, Sept 2000.
31. B. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc. of 27th Int. Symp. on Microarchitecture*, pages 67–74, Nov 1994.
32. J. Sánchez and A. González. Cache Sensitive Modulo Scheduling. In *Proc. of 30th Int. Symp. on Microarchitecture*, pages 338–348, Dec 1997.
33. J. Sánchez and A. González. Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture. In *Proc. of the 33rd Int. Symposium on Microarchitecture*, pages 124–133, Dec 2000.
34. J. Sánchez and A. González. The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures. In *Procs. of the Int. Conf. on Parallel Processing (ICPP'00)*, pages 555–562, August 2000.
35. J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Improved Spill Code Generation for Software Pipelined Loops. In *Procs. of the Programming Languages Design and Implementation (PLDI'00)*, June 2000.
36. J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures. In *Proc. of the 34th Int. Symp. on Microarchitecture*, December 2001.

Towards a Proposal for the Formalization of Ideal Rational Agents

J.C. Burguillo Rial, M.J. Fernández Iglesias, M. Llamas Nistal

Departamento de Ingeniería Telemática
jrial@det.uvigo.es,
WWW: http://www-gist.det.uvigo.es
ETS de Ingenieros de Telecomunicación
Campus Universitario S/N
36200 Vigo

Resumen Parece evidente que los sistemas o sociedades de agentes son un enfoque adecuado para la descripción de sistemas concurrentes. Debido a la complejidad de los sistemas distribuidos y en particular de los sistemas multiagente, parece conveniente disponer de un marco formal apropiado para estudiar sus características, y así facilitar su desarrollo y validación. Tradicionalmente, el tratamiento teórico del concepto de agente se ha basado en las teorías de formalización del conocimiento provenientes del campo de la Inteligencia Artificial. Nosotros pensamos que también es posible enfocar este problema desde el punto de vista de la formalización de los sistemas concurrentes, donde ya existe una base teórica robusta y bien establecida. En esta línea, proponemos una teoría para la especificación formal de agentes que extiende de manera natural resultados clásicos en el campo de los sistemas concurrentes. Este artículo presenta los primeros resultados.

1 Introduction

The concept of *agent* is becoming more and more important in both Artificial Intelligence and widespread Computer Science. The agent oriented approach to design is based on decomposing problems in terms of autonomous agents that exhibit proactive and intelligent behaviour, and interact with each other in terms of higher-level protocols and languages.

When the agent model is applied to distributed system design, *multi-agent systems* or *agent societies* appear. Multi-agent systems are ensembles of agents, acting independently from each other (recall that agents have intelligent and proactive behaviour) to accomplish their own tasks. For this, they have to interact with other agents and with their environment to gather the information or services they need. Additionally, an agent has to coordinate its activities with other agents in the society to ensure that overall goals can be met.

Thus, a multi-agent system will achieve its goals through agent collaboration. In many cases, an agent society achieves more complex goals than the mere aggregation of the individual agent's goals. For example [20], in a system composed

of autonomous agents looking for computational resources, a global load balancing can be achieved, despite the fact that no agent explicitly worries about load balancing.

It has been shown that there are strong arguments in favor of agent-based software engineering [5]. For complex systems, agent-oriented decomposition is an effective way of partitioning the problem space, the key abstractions of the agent-oriented design attitude provide a natural approach to modelling, and agent orientation provides an appropriate philosophy for dealing with organisational relationships.

In this paper we propose a formal model for agents that extends well-established results in the field of concurrent and distributed systems to take into account the defining properties of agency. This model is constructed to be particularly adequate for the verification and validation of multi-agent systems, and is based on a grey box [9] approach to system modelling.

Grey box models [2] are present as a situation when the structure of the system to be verified is known, but the implementation details within each component remain hidden. In our case, this approach will support verification and validation within agent societies, where agents communicate by means of clearly defined interfaces. Each agent in a multi-agent system may be observed according to two different perspectives [20]: from the inside, as an individual software system, and from the outside, as part of a society, interacting with other individuals, accessing resources and exploiting the social infrastructure. Using grey box models, agent interaction can be studied abstracting apart nonrelevant intra-agent details.

The rest of the paper is organized as follows. The next section discusses some preliminary concepts and notation that are used along the paper. Section 3 proposes a formal model for agents as Agent Oriented Transition Systems, an extension of classical Labelled Transition Systems. Section 4 further elaborates the formal model proposed and introduces some useful relations based on it. Section 5 introduces multi-agent systems in this framework. Finally, section 6 offers a brief discussion about this contribution.

2 Preliminary Concepts and Notation

Along the next paragraphs we present basic theoretical concepts that serve as the foundation of our proposal. First, we introduce Labelled Transition Systems, which is a basic model that we shall extend to represent agents, and more specifically, the intra-agent view. This model is simple, theoretically sound, and will support a straightforward representation of intra-agent details needed to specify inter-agent communication. Then, we discuss the relevant relations among Labelled Transition Systems that will support verification and validation.

2.1 Labelled Transition Systems

Labelled Transition System (LTS) is a basic model to describe the behaviour of processes. In this proposal it will be used to represent the basic behaviour of agents.

Definition 1 *A labelled transition system is a 4-tuple* $< Stat, L, T, s_0 >$ *where:*

- $Stat$ *is a countable, non-empty set of states;*
- L *is a countable set of labels;*
- $T \subseteq Stat \times (L \cup \{i\}) \times Stat$ *is the countable set of transitions. Label (i) denotes a special internal action, referred as* (τ) *in some models [8];*
- $s_0 \in Stat$ *is the initial state.*

We denote the class of all labelled transition systems over L by $LTS(L)$. We restrict $LTS(L)$ to systems that are strongly convergent, i.e., those that do not have infinite compositions of transitions with internal actions. We also assume in the sequel that the sets $Stat, L$ and T are finite. LTS may model the semantics of formal languages as LOTOS [4], CSP [1], CCS [8].

Definition 2 *Let* $P =< Stat, L, T, s_0 >$ *be a labelled transition system with* $s, s' \in Stat$. *We introduce the following notation:*

Notation	**Meaning**
L	is the alphabet of observable actions, $a, b, c, ...$ (except i) will denote its elements.
L^*	is the set of traces that may be composed with actions of L; $\sigma, \sigma', \sigma_j$ will denote its elements.
$s - \mu \rightarrow s'$	where $\mu \in L \cup \{i\}$, represents the transition relation.
$s = \epsilon \Rightarrow s'$	$s \equiv s'$ or $s - i^n \rightarrow s'$ for $n \geq 1$.
$s = a \Rightarrow s'$	$\exists s_1, s_2 \in Stat$ and $s = \epsilon \Rightarrow s_1 - a \rightarrow s_2 = \epsilon \Rightarrow s'$.
$s = \sigma \Rightarrow s'$	Exists a set $\{s_1, ..., s_{n-1}\}$ and $\sigma = a_1 \dots a_n$, such that $s = a_1 \Rightarrow s_1 = \cdots \Rightarrow s_{n-1} = a_n \Rightarrow s'$.
$s = \sigma \Rightarrow$	$\exists s' \in Stat$ such that $s = \sigma \Rightarrow s'$.
$s \neq \sigma \Rightarrow$	There is no $s' \in Stat$ such that $s = \sigma \Rightarrow s'$
Tr(P)	$\{\sigma \in L^* \mid P = \sigma \Rightarrow\}$, the set of traces accepted by P.
Init(P)	$\{a \in L \mid P = a \Rightarrow\}$, the set of L labels accepted by P.
P **after** σ	$\{s' \mid P = \sigma \Rightarrow s'\}$, the set of states reached from P by the trace σ.

The special label $i \notin L$ represents an unobservable, internal action. A trace $\sigma \in L^*$ is a finite sequence of observable actions over L with ϵ denoting the empty sequence. The trace $\sigma_1.\sigma_2$ is the concatenation of σ_1 and σ_2. We say that σ_1 is a prefix of σ if and only if there is a trace σ' such that $\sigma = \sigma_1.\sigma'$, and we denote it by $\sigma_1 \preceq \sigma$. We use the notation $a \lll \sigma$ if label a appears in trace σ. We extend and overload this notation, and write $\sigma' \lll \sigma$ when trace σ' is inserted

in σ, but with other actions in between. If it is clear from the context, we will represent an LTS by its initial state. For example, if $P =< Stat, L, T, s_0 >$, we use $P = \sigma \Rightarrow$ or $s_0 = \sigma \Rightarrow$ to say that (the initial state of) P accepts trace σ.

We represent an LTS by a tree or a graph, where nodes represent states and edges represent transitions, or by a process-algebraic behaviour expression, whose operational semantics are given by the axioms and inference rules that appear in table 1. In that table, the first four entries provide the basic operators to construct an LTS. Inaction represents a final state, action prefix and internal actions are used to construct transitions between states, and choice provides branching. Synchronization provides representation of inter-LTS interaction, and full synchronization is a particular case of synchronization where $G = L$. In this case, we use an alternate compact notation (i.e. $||$) and we provide again the relevant derivation rules for the sake of clarity.

Name	Syntax	Axioms and Derivation Rules
Inaction	**stop**	
Action Prefix	$a;B$	$a;B - a \rightarrow B$
Internal Actions	$i;B$	$i;B - i \rightarrow B$
Choice	$B_1[]B_2$	$\frac{B_1 - \mu \rightarrow B_1'}{B_1[]B_2 - \mu \rightarrow B_1'}$ $\quad$ $\frac{B_2 - \mu \rightarrow B_2'}{B_1[]B_2 - \mu \rightarrow B_2'}$
Synchronization: $G \subseteq L$	$B_1\|[G]\|B_2$	$\frac{B_1 - \mu \rightarrow B_1', \mu \notin G}{B_1\|[G]\|B_2 - \mu \rightarrow B_1'\|[G]\|B_2}$ $\quad$ $\frac{B_2 - \mu \rightarrow B_2', \mu \notin G}{B_1\|[G]\|B_2 - \mu \rightarrow B_1\|[G]\|B_2'}$ $\quad$ $\frac{B_1 - \mu \rightarrow B_1', B_2 - \mu \rightarrow B_2', \mu \in G}{B_1\|[G]\|B_2 - \mu \rightarrow B_1'\|[G]\|B_2'}$
Full Synchronization	$B_1\|\|B_2$	$\frac{B_1 - i \rightarrow B_1'}{B_1\|\|B_2 - i \rightarrow B_1'\|\|B_2}$ $\quad$ $\frac{B_2 - i \rightarrow B_2'}{B_1\|\|B_2 - i \rightarrow B_1\|\|B_2'}$ $\quad$ $\frac{B_1 - a \rightarrow B_1', B_2 - a \rightarrow B_2', a \in L}{B_1\|\|B_2 - a \rightarrow B_1'\|\|B_2'}$

Tabla1. Syntax and Semantics

2.2 Relations among Labelled Transition Systems

To compare different transition systems, several relations have been proposed in the theory of concurrent and distributed programming. Specifically, in the context of LTS there have been several proposals like bisimulation equivalence [11], observational equivalence [7, 8], or testing equivalence [10], among others. A detailed analysis and comparison of these equivalences and many others can be found in [17, 18].

The observational equivalence or weak bisimulation is defined [8] as a relation among states of different transition systems. Informally, two systems $P, Q \in LTS(L)$ are observationaly equivalent if every state reached from P after executing a trace is observationaly equivalent to any state of Q, which

may be reached using the same trace; and this also holds if we interchange P and Q. This relation intuitively captures the notion of externally indistinguible behaviour among systems.

Instead of comparing behaviours in an intensional way, that is, comparing their states and transitions, it is also possible to relate the behaviour of systems by an extensional characterization, that is, by experimentation. In [10] appears, for the first time, a relation to model the behaviour of systems using experiments and compare the observations obtained. Three types of experiments are defined for a process: those ones whose executions always fail, those one whose executions are sometimes successful, and the ones that are always successful. For the last two types of experiments, Nicola and Hennessy define two different semantics, denoted by **may** and **must** semantics. A process may pass an experiment T (P **may** T) if their composition ($P||T$) has, at least, a successful execution, while P must pass T (P **must** T) if all possible executions of ($P||T$) are successful. Combining these two semantics, they obtain a third type of semantics: **may** − **must**. Two processes are **may** − **must** equivalent if and only if they pass the same experiments. Besides these equivalences, we may obtain partial preorders among processes. For example, we say that P is *better* than Q if any experiment successfully passed by Q is also passed by P:

$$Q \sqsubseteq_{\mathbf{must}} P \Leftrightarrow \forall T \ (Q \ \mathbf{must} \ T \Rightarrow P \ \mathbf{must} \ T)$$
$$Q \sqsubseteq_{\mathbf{may}} P \Leftrightarrow \forall T \ (Q \ \mathbf{may} \ T \Rightarrow P \ \mathbf{may} \ T)$$

In the general case[15], for a set of experiments Set_T, and a set of observations $\mathbf{Obs}(T, P)$ obtained through experiment $T \in Set_T$ over system P, we define relations between two systems to compare the observations $\mathbf{Obs}(T, P)$ against $\mathbf{Obs}(T, Q)$. Formally:

$$P \ \mathbf{rel} \ Q \Leftrightarrow_{def} \forall T \in Set_T : \mathbf{Obs}(T, P) * \mathbf{Obs}(T, Q)$$

By varying the class of experiments Set_T, the observations **Obs** that an observer can make, and the relation * between observations of P and Q, many different relations may be defined. Now we introduce two classical ones:

- *Trace Preorder ($\leq_{tr}$).* Let $P, Q \in LTS(L)$. We have:

  $$Q \leq_{tr} P \Leftrightarrow_{def} \mathbf{Tr}(Q) \subseteq \mathbf{Tr}(P) \Leftrightarrow \mathbf{Obs}_t(Q) \subseteq \mathbf{Obs}_t(P)$$

  where $\mathbf{Obs}_t(T, P) =_{def} \{\sigma \in L^* \mid (T||P) = \sigma \Rightarrow\}$. Intuitively, $Q \leq_{tr} P$ when process Q only shows part of the behaviour in P, in terms of traces of observable actions.
- *Testing Preorder ($\leq_{te}$).* Let $P, Q \in LTS(L)$. We say that $Q \leq_{te} P$ if and only if

  $$\forall T \in LTS(L) : \mathbf{Obs}_c(T, Q) \subseteq \mathbf{Obs}_c(T, P) \text{ and } \mathbf{Obs}_t(T, Q) \subseteq \mathbf{Obs}_t(T, P)$$

  Where $\mathbf{Obs}_c$ and $\mathbf{Obs}_t$ are respectively the observations obtained over the complete traces (deadlock traces) and over the normal traces. Formally:

$$\mathbf{Obs}_c(T,P) =_{def} \{\sigma \in L^* \mid L \in \mathbf{Ref}((T||P),\sigma)\}$$
$$\mathbf{Obs}_t(T,P) =_{def} \{\sigma \in L^* \mid (T||P) = \sigma \Rightarrow\}$$

This relation is obtained when we select transition systems as observers (**Obs**), the observation relation (*) is the set inclusion, and the obtained observations are the traces. Then, its intensional characterization will be:

$$Q \leq_{te} P \Leftrightarrow \forall \sigma \in L^* : \mathbf{Ref}(Q,\sigma) \subseteq \mathbf{Ref}(P,\sigma)$$

Using these preorders we can obtain the induced equivalences in a natural way:

- *Trace Equivalence ($\approx_{tr}$).* Let $P, Q \in LTS(L)$. We have:

$$Q \approx_{tr} P \Leftrightarrow_{def} Q \leq_{tr} P \text{ and } P \leq_{tr} Q$$

  Therefore, if $Q \approx_{tr} P$ then $\mathbf{Tr}(Q) = \mathbf{Tr}(P)$, i.e., $\mathbf{Obs}_t(Q) = \mathbf{Obs}_t(P)$.
- *Testing Equivalence ($\approx_{te}$).* Let $P, Q \in LTS(L)$. We have:

$$Q \approx_{te} P \Leftrightarrow_{def} Q \leq_{te} P \text{ and } P \leq_{te} Q$$

  Therefore $Q \approx_{te}$ if and only if

$$\forall T \in LTS(L) : \mathbf{Obs}_c(T,Q) = \mathbf{Obs}_c(T,P) \text{ and } \mathbf{Obs}_t(T,Q) = \mathbf{Obs}_t(T,P)$$

  And the corresponding intensional characterization will be:

$$Q \approx_{te} P \Leftrightarrow \forall \sigma \in L^*, \mathbf{Ref}(Q,\sigma) = \mathbf{Ref}(P,\sigma)$$

3 Formalization of an Agent System

As an initial approach, an agent is anything that perceives its environment through *sensors* and acts through *effectors* [13]. Software agents are programs that have the following properties [19]:

- *Autonomy*: Agents operate without the direct intervention of other entities, and have some control over their actions and state.
- *Social ability*: Agents interact with other entities like other agents, legacy software, or humans.
- *Reactivity*: Agents perceive their environment and respond to changes in it.
- *Proactiveness*: They are able to exhibit goal-directed behaviour by taking the initiative.

An ideal rational agent is an agent that, for each possible percept sequence, should do whatever action is expected to maximice its performance measure (rational), on the basis of evidence provided by the percept sequence and whatever built-in knowledge the agent has [13].

>From the definition above, it seems convenient to describe agents by its intentional stance, taking into account its information attitudes (belief, knowledge) and pro-attitudes (desire, intention, obligation, etc.). Most theories of agency follow this approach (see [19] for a survey). However, although it is generally accepted that at least one information attitude and one pro-attitude is needed to adequately describe agents, there is no clear consensus about which combinations are best suited to characterize rational agents.

For the model described in this paper, both kinds of attitudes correspond to *events*. Let L be a set of events. We define the following subsets from L:

- $L_P \subset L$ is the set of *percepts*, that is, events reaching an agent through its sensors.
- $L_A \subset L$ is the set of *actions*, that is, events leaving the agent through its effectors.
- $L_D \subset L$ is the set of *desires*, that is, events that may become percepts. We will assume that our agent has common sense concerning its desires and therefore $L_D \subseteq L_P$.
- $L_I \subset L$ is the set of *intentions*, that is, events that may become actions. Again, we assume agents who know their limitations, and therefore $L_I \subseteq L_A$.

As it is stated in [14], communication between a process and its environment, both modelled as LTS (c.f. section 2.1), is based on a symmetric interaction, as expressed by the composition operator ($||$). An interaction can occur if both, the process and its environment, are able to perform that interaction, implying that its occurrence may be blocked by any of them, even by both. If the process and its environment offer more than one interaction, then it is assumed that, by some mysterious negotiation mechanism, they will agree on a common one. There is no notion of input or output, nor of initiative or direction. All actions are treated in the same way for both communicating partners.

Agents, however, communicate in a different manner. They make a distinction between inputs and outputs, and one can clearly distinguish whether the initiative for a particular interaction is with the system or with its environment. There is a direction in the flow of information from the initiating process. This initiating process determines which interaction will take place.

We will now consider a class of (models of) processes to describe ideal agents for which the set of actions L can be partitioned into outputs (actions performed through the effectors), for which the initiative to perform them is with the agent, and inputs (percepts from the sensors), for which the initiative is with the environment.

If an input is initiated by the environment, the agent is always prepared to participate in such interaction: all inputs in the agent are always enabled; they can never be refused. The communication is still synchronous: if an interaction occurs, it occurs at exactly the same time in both processes. The communication, however is not symmetric.

This model of processes strongly resembles the Input/Output Transition Systems described in [14, 15], which is also analogous to Input/Output Automata

[6] and Input/State Machines [12]. Additionally, we introduce a set of functions to describe agent properties.

Definition 3 (Agent Oriented Transition System (AOTS)) *An AOTS is a 7-tuple* $< Stat, L_P, L_A, T, D, I, U, s_0 >$, *where* $< Stat, L_P, L_A, T, s_0 >$ *is an LTS whose set of events L is partitioned into perceptions* L_P *and actions* L_A ($L_P \bigcup L_A = L$ *and* $L_P \bigcap L_A = \emptyset$), *and for which all perceptions in* L_P *are always enabled in any state:* $\forall \sigma \in L^*$, $\forall Q' \in Q$ **after** σ, $L_P \in$ **Init**(Q').

Besides, an AOTS has three functions:

- *The desires function:* $D : Stat \times L^* \rightarrow L_D^*$ *represents agent desires. Given a initial set of desires* $D(s_0, \epsilon)$, *the set of desires at any state s will be* $D(s, \sigma) = D(s_0, \epsilon)/\sigma$ *where* $\sigma \in L^*$ *is the trace followed by the agent to reach state s (execution trace), and* $D(s_0, \epsilon)/\sigma$ *means that we remove from the initial desires (set of desired perceptions) those ones already fulfilled by* σ.
- *The intentions function:* $I : Stat \times L^* \rightarrow L_I^*$ *represents agent intentions. Given a set intentions* $I(s_0, \epsilon)$ *the agent had at the initial state, the set of intentions at any state s will be* $I(s, \sigma) = I(s_0, \epsilon)/\sigma$; *meaning that we remove from the initial intentions (trace of actions belonging to* L_A*) those ones already fulfilled by* σ.
- *The utility function:* $U : T \rightarrow \mathbb{R}$ *represents the utility that any transition has for the agent and helps the agent to decide the action to perform next. This function may evolve during the agent "live" to incorporate the agent learning.*

The class of Agent Oriented Transition Systems is denoted by $AOTS(L_P, L_A)$. We use suffixes (p, a) to indicate whether an action belongs respectively to the set of perceptions L_P or to the set of actions L_A.

4 Agent Relations

In this section we discuss some relations to compare agents described using the AOTS formalism. We begin our relation description considering a new version of **may** – **must** semantics for AOTS following the approach from Nicola and Hennessy introduced in section 2.2.

$$Q \sqsubseteq_{\mathbf{must}} P \Leftrightarrow \forall T \in Set_I \ (Q \ \mathbf{must} \ T \Rightarrow P \ \mathbf{must} \ T)$$
$$Q \sqsubseteq_{\mathbf{may}} P \Leftrightarrow \forall T \in Set_I \ (Q \ \mathbf{may} \ T \Rightarrow P \ \mathbf{may} \ T)$$

where $Set_I \subseteq L_I^*$ is the set of all possible intentions an agent have.

Now, we introduce some additional relations:

- *Action Preorder* ($\leq_{atr}$). Let $P, Q \in AOTS(L_P, L_A)$ we have:

$$Q \leq_{atr} P \Leftrightarrow_{def} \mathbf{aTr}(Q) \subseteq \mathbf{aTr}(P)$$

To formally define $\mathbf{aTr}(P)$, we need to introduce *projections* first. A projection $\mathbf{TrPr}(\sigma, L_A) = \sigma\lceil L_A$ is a trace that fulfill the next rules:

$$\text{- } \epsilon\lceil L_A =_{def} \epsilon$$
$$\text{- } (a.\sigma)\lceil L_A =_{def} \begin{cases} a.(\sigma\lceil L_A) & if\ a \in L_A \\ \sigma\lceil L_A & if\ a \notin L_A \end{cases}$$

Using projections we can define the action trace set $\mathbf{aTr}(P)$:

$$\mathbf{aTr}(P) = \{\sigma_a \in L*_A \mid \sigma_a = \mathbf{TrPr}(\sigma, L_A) \text{ for } \sigma \in \mathbf{Tr}(P)\}$$

which means that $\mathbf{aTr}(P)$ contains all projections of P traces over the set of actions L_A.
Intuitively, $Q \leq_{atr} P$ if agent Q only shows part of the behaviour, in terms of traces of observable actions, which appears in agent P. In this preorder we only consider actions performed in traces, and we discard perceptions.

– *Agent Oriented Trace Preorder ($\leq_{aotr}$).* Let $P, Q \in AOTS(L_P, L_A)$. We have:

$$Q \leq_{aotr} P \Leftrightarrow_{def} \mathbf{Tr}(Q) \subseteq \mathbf{Tr}(P)$$

– *Agent Oriented Testing Preorder ($\leq_{aote}$).* Let $P, Q \in LTS(L)$. We have:

$$Q \leq_{aote} P \Leftrightarrow_{def} \mathbf{Tr}(Q) \subseteq \mathbf{Tr}(P) \text{ and } \mathbf{asTr}(Q) \subseteq \mathbf{asTr}(P)$$

where $\mathbf{asTr}(P)$ are the *action suspension traces* of agent P. They are traces that drive P to states where it can not proceed autonomously, without perceptions from its environment, that is, a state where no external or internal actions are possible.

Definition 4 (State Actions Set) . *Let p be a state in an AOTS and P be a set of states; then*

1. $\mathbf{act}(p) =_{def} \{a \in L_A \mid p - a \rightarrow\}$
2. $\mathbf{act}(P) =_{def} \bigcup\{\mathbf{act}(p) \mid p \in P\}$

Now, we can easly define the corresponding intensional characterization. Let $P, Q \in LTS(L)$:

$$Q \leq_{aote} P \Leftrightarrow_{def} \forall \sigma \in L^* : \mathbf{act}(Q \textbf{ after } \sigma) \subseteq \mathbf{act}(Q \textbf{ after } \sigma)$$

This preorder has been derived from the input-output testing preorder $\leq_{iote}$ described in [15], where the reader may check the correctness of the definitions.

Using these preorders we can obtain the induced equivalences in the same way we described in section 2.2.

5 Multi-Agent Systems

In this section we study systems composed by several agents that interact, where interactions can be made explicit as cooperation, negotiation, competition, and mainly as communication.

5.1 Modeling Communication

Unfortunately, AOTSs are not so good when it is necessary to achieve agent communication in a concurrent system. Suppose we have agents P and Q in figure 1.

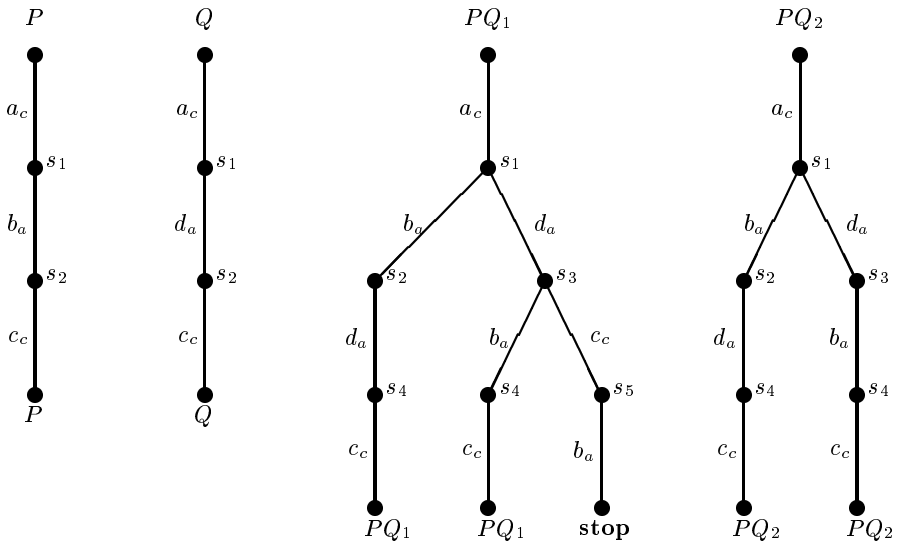


Figura1. P, Q, $PQ_1 = P|[a_c, c_c]|Q$ with perceptions/actions and $PQ_2 = P|[a_c, c_c]|Q$ with perceptions/actions/communications.

First, suppose that we want to model communication between agents P and Q using the previous perceptions/actions scheme. In such framework, agent $P \in AOTS(\{c_c\}, \{a_c, b_a\})$ desires to send a message to agent Q through action a_c, then it executes any action b_a and finally waits to receive a message from agent $Q \in AOTS(\{a_c\}, \{d_a, c_c\})$. Agent Q waits to receive perception a_c from agent P, then it performs action d_a and finally informs agent P through action c_c. If we compose these two agents considering their common perceptions and actions, we obtain the system PQ_1. In that system, agents P and Q communicate thorugh actions $\{a_c, c_c\}$, which are perceptions or actions depending on the agent. The other actions $\{b_a, d_a\}$ are in both agents actions oriented towards the environment. The system PQ_1 shows that there is a deadlock situation when agent Q performs d_a before agent P performs b_a. Thus, execution races appear due to concurrence.

These problems could be solved following two different approaches. On the one side, we can use FIFO queues to communicate among agents. This approach has been already studied in the area of distributed systems for software testing

[16], and drives to complex descriptions and relations. In fact, this situation may be perfectly modeled by IOTSs, and therefore by AOTSs (c.f [14]).

The second approach is to model such communication using symmetrical synchronous interactions, as it was done with classical LTSs. This approach has also been introduced in figure 1 using again $\{a_c, c_c\}$ to communicate both agents. But now, $\{c_c\}$ is not a perception of P and $\{a_c\}$ is not a perception of Q. The result is the system PQ_2, which now does not experience any undesirable deadlock.

We formalize the model to describe such communications in the next paragraphs.

5.2 Multi-Agent Oriented Transition Systems

In this section we introduce and define multi-agent oriented transition system to model agent societies maintaining all information related to the agent or agents that perform any event at any given state.

Definition 5 (Multi-Agent Oriented Transition System (MAOTS))
Let $A^k \in AOTS(L)$, $k = 1..N$ be a set of agents. A multi-agent oriented transition system (MAOTS) is a 8-tuple $< Stat, ML_P, ML_A, ML_C, T, D, I, U, s_0 >$, which can be considered as an AOTS whose set of events ML has been partitioned into percepts ML_P, actions ML_A and agent communications ML_C ($ML_P \bigcup ML_A \bigcup ML_C = ML$ and ML_P, ML_A and ML_C are all disjoint).

Besides, the set ML_P (respectively, ML_A or ML_C) is a countable set of indexed labels (a, V), where $a \in L_P$ (L_A or L_C) and $V \in PowerSet(\{1, .., N\})$. Again, all percepts in ML_P are always enabled in any state.

The class of Multi-Agent Oriented Transition Systems is denoted by the set MAOTS(ML_P, ML_A, ML_C). We use suffix (c) to indicate whether an event belongs to the set of agent communications L_C. Due to this new set of indexed labels, we need to redefine the parallel operator:

Definition 6 (Agent Oriented Parallel Operator: $||_{ao}$)
Let $P, Q \in MAOTS(ML)$, $G \subseteq ML_C$, and $\mu \in ML \bigcup \{i\}$. We define $P|[G]|_{ao}Q$ as:

1. $P - (\mu, V) \rightarrow P'$, $(\mu \notin G, V \in P(\{1, .., N\}))$ **then** $P|[G]|_{ao}Q - (\mu, V) \rightarrow P'|[G]|_{ao}Q$
2. $Q - (\mu, W) \rightarrow Q'$, $(\mu \notin G, W \in P(\{1, .., N\}))$ **then** $P|[G]|_{ao}Q - (\mu, W) \rightarrow P|[G]|_{ao}Q'$
3. $P - (\mu, V) \rightarrow P', Q - (\mu, W) \rightarrow Q'$, $(\mu \in G$, *the sets* $V, W \in PowerSet(\{1, .., N\})$ **then** $P|[G]|_{ao}Q - (\mu, V \bigcup W) \rightarrow P'|[G]|_{ao}Q'$

This parallel operator is used to specify the composition of single agents, keeping track of the origin of all events through indexes. It can also be recursively applied to the composition of MAOTS that stem from the composition of simpler ones. In other words, transition labels may have more than one index.

Using this model, when a composed system is created from simpler parts, we can keep track of the original agents participating in every single event. For example, if we have agents that appear in figure 2. Their composition maintains an index per transition to indicate what agents interact in the obtained MAOTSs.

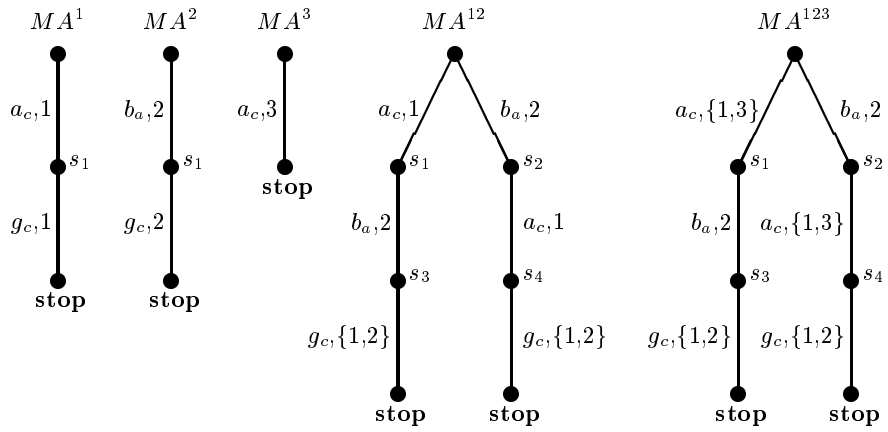


Figura2. Agents MA^1, MA^2 and MA^3. Multi-Agent System $MA^{12} = MA^1 |[g_c]|_{ao} MA^2$ and Multi-Agent System $MA^{123} = MA^{12} |[a_c]|_{ao} MA^3$

6 Discussion

We have introduced the foundations for a theory of agency based on sound, well established results in the field of concurrent and distributed programming [2]. Basically, in this paper we provide a representation model for agents and multi-agent systems, and show that this model is a good candidate to support the formal development of agent systems. Obviously, these preliminary results have to be extended to support the formal development of multi-agent systems. In this line, we are working on techniques and tools to effectively support the verification, testing and validation of agents and agent societies.

There are several properties of agency that an hypothetical model to describe agent systems and multi-agent communities has to consider. Next, we summarize some of them, stablishing at the same time how to describe them using AOTSs and MAOTSs.

- **Naming**: In a multi-agent system every agent needs a unique name to be identified to successfully interact with other agents, and to allow a coordinated distribution of tasks. In our proposal, we use indexes in agent societies described as MAOTSs. Of course, in a practical implementation such indexes can be substituted by literal names.

- **Concurrency**: Multi-agent and distributed systems are concurrent by nature. As the model we have chosen to describe such systems comes from the distributed systems theory, the way to model concurrency, partial sincronization and interleaving using the parallel operator ($||_{ao}$) is very natural and has a sound and well established support.
- **Distribution**: Multi-agent systems comprise multiple elements, possibly distributed over a network. This situation may be modeled by the MAOTS in a very simple manner.
- **Communication**: Interactions among agents in a multi-agent system may adopt multiple forms. Among them, coordination of actions, cooperation in the resolution of tasks and competition for resources may need the exchange of messages among agents. For every agent, such communication messages have been modeled in two ways: interactions with the environment through a disjoint set of perceptions (always enabled), and actions (always accepted by the environment, but maybe not performed effectively). This enables a natural description of the evolution of agent behaviour. Communication with other agents may be also implemented using the same scheme. However, we also introduced classical LTS synchronization to support the formalization of synchronous and symmetric communications.
- **Agent Autonomy**: Autonomy is a key characteristic of agents. By autonomy we mean that an agent is able to act without intervention of humans or any other external support. It is directed by a set of desires or tendencies. The set of tendencies an agent has can be described easily using the function $D(s)$ and the set of states and actions to formalize future planning.
- **Decision-making and Knowledge Representation**: every agent has to make decisions about what and when to do something. Related with this agent capability is knowledge representation and agent learning. An agent can learn by its perceptions or by its interactions with other agents. Therefore, there is a need for external knowledge representation. Such knowledge representation has been considered in our model through the utility function $U(t)$, bound to every transition in the system. The way in which such utility function is calculated or modified by every agent is let open to use any type of decision-making or knowledge representation used by a specific agent design.

Along the last few years there has been some initiatives to study biological models constructed by reactive agents, which do not exhibit intelligent behaviour when isolated. Nevertheless, the aggregation of these reactive agents when forming colonies and societies show many of the properties that could be attributed to an intelligent entity [3]. This approach contrasts with that of cognitive agents and the proposals of the school of Distributed Artificial Intelligence.

Using MAOTSs, it is fairly simple to describe and combine multiple entities using the parallel operator to obtain a multi-agent system. This agent society will evolve according to the specified agents' reactive behaviour and the environment characteristics, both represented in the proposed framework.

Besides, a well established theory of relation-based formal analysis, validation and verification is already developed in the field of LTS, and ready to be applied to this new context.

Referencias

1. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31, 1984.
2. J.C. Burguillo. *Contribución a la Fase de Prueba de Sistemas Concurrentes y Distribuidos mediante Técnicas de Descripción Formal.* PhD thesis, Universidade de Vigo, 2001.
3. J. Ferber. *Multi-Agent Systems.* Addison-Wesley, 1999.
4. ISO. Lotos a formal description technique based on the temporal ordering of observational behaviour. International Standard, 1989.
5. Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.
6. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
7. R. Milner. Calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.
8. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
9. D. Morris and B. Tamm, editors. *Concise Encyclopedia of Software Engineering.* Pergamon Press, Oxford, 1993.
10. R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoret. Comp. Science*, 34:83–133, 1984.
11. D. Park. Concurrency and automata on infinite sequences. *Lecture Notes in Computer Science*, 104, 1981.
12. M. Phalippou. *Relations d'Implantation et Hypothéses de Test sur des Automates á Entrées et Sorties.* PhD thesis, Université de Bordeaux I, 1994.
13. Stuart Rusell and Peter Norvig. *Artificial Intelligence. A Modern Approach*, chapter 2, page 31. Prentice-Hall Inc., 1995.
14. J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.
15. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17:103–120, 1996.
16. J. Tretmans and L. Verhaar. A queue model relating synchronous and asynchronous communication. In *Protocol Specification, Testing and Verification XII*, pages 131–145, 1992.
17. R.J. van Glabbeek. The linear time-branching spectrum. *Lecture Notes in Computer Science*, 458:278–297, 1990.
18. R.J. van Glabbeek. The linear time-branching spectrum ii (the semantics of sequential systems with silent moves). *Lecture Notes in Computer Science*, 715:66–81, 1993.
19. Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Konwledge Engineering Review*, 10(2):115–152, 1995.
20. F. Zambonelli, N. R. Jennings, A. Omicini, and M. Wooldridge. Agent-oriented software engineering for internet applications. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Tecnhnologies and Applications*, chapter 13. Springer Verlag, 2000.

A Continuation-Based Model for a Distributed Functional Setting*

Mercedes Hidalgo-Herrero[1] and Yolanda Ortega-Mallén[2]

[1] Dept. Didáctica de las Matemáticas
Facultad de Educación, Universidad Complutense de Madrid, Spain
mhidalgo@edu.ucm.es
[2] Dept. Sistemas Informáticos y Programación
Facultad CC.Matemáticas, Universidad Complutense de Madrid, Spain
yolanda@sip.ucm.es

Abstract Combining distributed parallelism and laziness requires some relaxation of the latter on behalf of the former, with the consequent introduction of speculative computation, which will range between a minimum and a maximum in each particular program execution, due to the time-dependency inherent in parallel systems. Besides, dealing with side-effects (process creation and value communication) requires to take care of the order in which expressions are evaluated.
With these ideas in mind, we define a continuation-based denotational semantics for a very simple lazy language with explicit process creation and implicit process communication, which provides the basis for the formal study of other lazy functional parallel/concurrent languages like GPH [THM+96] or Eden [BLOP97].

1 Introduction

As the current state of art corroborates ([HM99]), there are many different ways to combine functional programming and parallelism. The advantages and drawbacks of all these approaches have already been discussed at length, thus we will not continue here with the comparison topic. It is the purpose of the present work to concentrate on a particular case: the adaptation of a lazy functional computation language for its execution in a distributed setting, and its formalization through the definition of a denotational semantics.

On the one hand, lazy evaluation is just an efficient implementation of normal order evaluation, where duplicated computations are avoided. More precisely, the arguments in a functional application are evaluated just once and only if they are ever needed. On the other hand, although it may be the case of having different kinds of processors designated for specific tasks, the usual purpose of distributing work is to speed-up computation. Therefore, a system where every task is strictly done under demand does not seem very promising. We would like

* Work partially supported by the Spanish project CICYT-TIC2000-0738 and the Spanish-British Acción Integrada HB1999-0102.

better to have a system where some amount of work is done in advance, so that the results are ready when they are needed. This means producing *speculative* computation, which may never be demanded. The exact amount of speculative parallelism will depend on the number of available processors, the scheduler decisions, the speed of basic instructions, etc. Hence, the execution of a program would range from reducing the speculation to the minimum – only what it is effectively demanded is computed – to expanding it to the maximum – every speculative computation is carried out –. While the former would be equivalent to executing the program on a single processor with a scheduler giving priority to the demand originated by the initial process, the latter would correspond to having an unlimited set of processors for evaluating the output of every generated process.

Another consideration to be taken concerns distributed memory. Under laziness and distribution conditions, processors cannot share a common closure heap and, in order to avoid frequent expensive communications, processes should be closed entities containing most of the information they need for their execution. For this purpose, a copy of closures from one process to another, is needed; for instance, a weak head normal form (whnf) must be communicated together with all the closures relative to its free variables.[1] A consequence of the copy of bindings is the possibility of duplicating work.

Looking for a denotational semantics to formally express program behaviour, we come to the conclusion that we are interested in three different aspects:

Functional : The final value produced by the program.
Parallel : The interactions between the processes which have been created during the computation.
Distribution : The degree of work duplication and speculation.

Few research has been carried out in this direction for parallel and/or concurrent functional languages. There has been some work addressing the first two aspects, for instance, in [DB97,FH99] two denotational semantics for the strict language Concurrent ML (CML) [Rep92] are presented. Both are based on the Acceptance Trees model [Hen88], originally defined for the analysis of reactive systems. In each paper, the model is extended with value production, in order to express the overall input-output relation of functional programs. The semantic model for strict evaluation is much simpler than one for lazy evaluation, and there is no need to bother about the *distribution* aspect mentioned above. As far as we know, ours is the first denotational semantics which considers this problem and expresses not only the final value, but also the interaction between processes during the computation in a lazy evaluation.

To detect work duplication and speculation, we want to express somehow the sharing/copying of closures, going with our approach just in the opposite

[1] As Abramsky points out in [Abr90], most functional compilers evaluate expressions to whnf values. In our calculus we follow this approach, and communicated values are also in whnf.

direction to the mentioned works: We depart from a *standard* denotational semantics [Sto77] expressing the input-output relation of functional programs and we extend it with a notion of *process system generation*. Maintaining the process system as part of the denotation of a program, means that the evaluation of an expression may produce some side-effects which must be treated with care in a lazy context. For instance, in the case of evaluating an application such as $(\lambda x.3)y$ in a process p, the evaluation of the variable y may imply the corresponding evaluation of other bindings. Whereas this is not relevant in the case of a denotational semantics which is only interested in the final value, in our approach the modifications in the process system should not be carried out, because the λ-abstraction is not strict in its argument and, therefore, the evaluation of y is not going to be demanded. Hence, process creation and value communication are side-effects which must take place only under demand. Moreover, we recall that laziness implies that arguments to function calls are evaluated at most once. Therefore, we must be careful to produce the corresponding side-effects only the first time a value is demanded.

In [Jos89] a continuation-based denotational semantics for the λ-calculus is given, reflecting the order in which expressions are evaluated and allowing for the description of *functional programming with side-effects* [Jos86]. Inspired by these works we obtain a continuation-based denotational semantics for a simple parallel lazy functional language.

2 The language

We will consider a lazy lambda calculus widened with a recursive `let` and a construction for creating parallel processes.

We need two syntactic categories: Identifiers ($x \in$ **Ide**) and expressions ($e \in$ **Exp**). The abstract syntax of the language is given in the following table:

$e ::=$	x		variable
$\mid$	$\backslash x.e$		λ-abstraction
$\mid$	$e_1 e_2$		application
$\mid$	$e_1 \texttt{\#} e_2$		process creation
$\mid$	$\texttt{let}\ \{x_i = e_i\}_n\ \texttt{in}\ e$	$(n > 0)$	local declaration

where $\backslash x.e$ stands for a functional abstraction (we reserve the λ symbol for the semantic functions) and $\texttt{let}\ \{x_i = e_i\}_n\ \texttt{in}\ e$ is an abbreviation for $\texttt{let}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$. This declaration construction is recursive, so that each local variable x_i can refer to every other in its definition e_i.

At execution time, an expression $e_1 \texttt{\#} e_2$ originates the creation of a process to evaluate the application expression $e_1 e_2$. Two channels are established between the new process and its creator: one for communicating the value of e_2 from the parent to the child, and a second one for communicating from the child to the parent the result value of the application. In $e_1 \texttt{\#} e_2$ the second expression, e_2, is evaluated by the parent resulting in a value v, while $e_1 v$ is evaluated by the child.

In a distributed setting there is no shared memory between processes, so that every communication between processes must take place through some channel. Therefore, all the definitions involved in the evaluation of e_1 are copied from the parent to the child at creation time. For the same reason, when communicating a functional abstraction, all the information relative to free variables in its body has to be copied (from the producer) to the consumer heap.

Functional abstractions as well as process definitions are expressed by lambda abstractions, but their applications lead to very different behaviours. While functional application is non-strict (the argument will be evaluated only if it is demanded by the body), in the presence of speculation there is an eager demand on the output produced by a process, thus implying eager process creation.

Let us explain in more detail this point through the following example, where we have introduced constant numbers to make it simpler.

Example 1. Laziness vs. eagerness.

```
let
  x = (\x1.\x2.x1)#2
  y = (\y1.3 * y1)#4
  z = (\z1.z1)8
in
  x z
```

Provided an infinite number of processors, the evaluation of this local declaration produces the following steps:

1. Introduction of the local variables x, y and z in the main process environment.
2. Creation of two new main process' children, one to apply the first λ-abstraction to 2, obtaining in this way the abstraction for the `let` body, and the other one to multiply 3 and 4.
3. Demand on the variable x.
4. Evaluation of the applications $(\backslash x_1.\backslash x_2.x_1)2$ and $(\backslash y_1.3 * y_1)4$ (although the latter has not been demanded, i.e. it is not needed for the evaluation of another expression) in the newly created processes.
5. Evaluation, in parallel with the evaluations in 4, of the application $(\backslash x_2.2)z$ in the main process, resulting the value 2.

It can be observed that z is not demanded by the functional application and, in concordance with the lazy evaluation, it is not evaluated. However, although y is neither demanded, a new process is created and its output expression is evaluated, because process creation is carried out when a #-expression is at top level, and both output and input evaluation are eager.

□

This overruling of laziness in favor of parallelism, and combined with the time-dependency inherent in parallel systems, introduces some non-determinism in the order of evaluation, which, even if it does not affect the final result in a transformational system, is crucial for the amount of speculative computation to be generated.

3 A Denotational Semantics

A goal of this semantics is to be able to recover the process system topology created during the evaluation of a program. But we are interested in this topology in a dynamic way, that is, we want to observe the interactions between the processes in the system, which are represented by communications. A set of channels in which each one is determined by the producer and the consumer processes, together with the communicated value, will allow us to obtain such information.

However, a process is more than what can be observed from the outside. Internally, a process is a set of bindings from variables (identifiers) to values. These values can either

- be expressed as closures, i.e. representing unevaluated expressions, or
- be under evaluation, and thus they cannot be shared, or
- be completely evaluated, i.e. in weak head normal form, or
- represent communications, i.e. indicating that the associated value has not been produced yet and, consequently, it has not been sent through the corresponding channel.

This aspect of a process will be represented as part of a general environment, from identifiers to values, broken up into parallel processes.

Therefore, a system will be described by these two components: a distributed environment and a channel set. The initial environment contains all the identifiers bound to undefined. Similarly, all the channels are initialized to free.

In the following, we will use $\oplus$ to express the extension/update of environments and channel sets, like for instance in $\rho \oplus \{(p, x) \mapsto \nu\}$. The operator $\oplus$ is also component-wise extended to systems.

Process creation and value communication can be considered as side-effects of the program evaluation, which modify the state of the system. To deal with these side-effects in our semantics, we resort to continuations, which express how the computation of a program should *continue* from a given state of the system.

The semantic domains that will shape up our denotational semantics are defined in Fig. 1.

Notice how the expressed values, or whnf values, are represented by the domain **EVal** which, in our calculus, just contains lambda abstractions, i.e. functions from identifiers to closures. Even though a closure is just represented by an expression, a new domain (**Clo**) is introduced to remark the fact that they are different semantic entities. Closures also differ from communications (**Comm**) in that the latter are qualified by the channel through which the value will be sent. The special value not_ready is introduced when a closure starts to evaluate and it will disappear at the end of that evaluation.

In the next sections, for the definition of the semantic functions, several auxiliary functions are used to obtain free identifiers:

$\mathbf{Cont}$	$= \mathbf{Sys} \to \mathbf{Sys}$	continuations
$\kappa \in \mathbf{ECont}$	$= \mathbf{EVal} \to \mathbf{Cont}$	expression continuations
$\Sigma \in \mathbf{Sys}$	$= \mathbf{Env} \times \mathbf{SChan}$	systems
$\rho \in \mathbf{Env}$	$= (\mathbf{IdProc} \times \mathbf{Ide}) \to (\mathbf{Val} + \{\mathsf{undefined}\})$	environments
$\delta \in \mathbf{SChan}$	$= \mathbf{IdChan} \to (\mathbf{Chan} + \{\mathsf{free}\})$	channel sets
$\mathbf{Chan}$	$= \mathbf{IdProc} \times \mathbf{ChVal} \times \mathbf{IdProc}$	channels
$\omega \in \mathbf{Val}$	$= \mathbf{Clo} + \{\mathsf{not\_ready}\} + \mathbf{EVal} + \mathbf{Comm}$	values
$\mu \in \mathbf{ChVal}$	$= \mathbf{EVal} + \{\mathsf{unsent}\}$	channel values
$\varepsilon \in \mathbf{EVal}$	$= \mathbf{Ide} \to \mathbf{Clo}$	expressed values
$\nu \in \mathbf{Clo}$	$= \mathbf{Exp}$	closures
$\mathbf{Comm}$	$= \mathbf{IdChan} \times \mathbf{Clo}$	communications
$x, y, z \in \mathbf{Ide}$		identifiers
$p, q \in \mathbf{IdProc}$		process identifiers
$ch \in \mathbf{IdChan}$		channel identifiers

Figure1. Semantic domains

$newIde :: \mathbf{Env} \to \mathbf{IdProc} \to \mathbf{Ide}$ $\rho(p, (newIde\ \rho\ p)) = \mathsf{undefined}$	$newIdChan :: \mathbf{SChan} \to \mathbf{IdChan}$ $\delta(newIdChan(\delta)) = \mathsf{free}$
$newIdProc :: \mathbf{Sys} \to \mathbf{IdProc}$ $p = newIdProc(\rho, \delta)$ $\forall (q, x) \in dom(\rho).(p \neq q)$ $\forall (q_1, \mu, q_2) \in rg(\delta).(p \neq q_1 \wedge p \neq q_2)$	$twoNewIdChans :: \mathbf{SChan} \to \{\mathbf{IdChan}\}$ $\{ch_1, ch_2\} = twoNewIdChans\ \delta$ $\forall i \in \{1, 2\}.\delta(ch_i) = \mathsf{free}$ $ch_1 \neq ch_2$

where for any function f, $dom(f)$ and $rg(f)$ correspond to the domain and the range of f, respectively.

In addition, in case of error the system remains unmodified, using for remarking that fact the continuation *wrong* (where $\forall \Sigma \in \mathbf{Sys}\,.(wrong\ \Sigma = \Sigma)$).

3.1 Lazy evaluation

The evaluation of an expression is done in a given specific process. Moreover, an expression continuation, modelling the rest of the computation once the expression has been reduced to weak head normal form, must be supplied. Therefore, we define

$$\mathcal{E} : \mathbf{Exp} \to \mathbf{IdProc} \to \mathbf{ECont} \to \mathbf{Cont}.$$

In order to obtain the final value and system after evaluating an expression e, $(p, main) \mapsto e$ is introduced in the initial environment, ρ_0, and the valuation would be $\mathcal{E}[\![main]\!]p\,\kappa\,\Sigma_0$.

Actually, this valuation function $\mathcal{E}$ will be given two definitions in order to express the minimal, or less speculative, semantics (valuation function $\mathcal{E}_{min}$) and the maximal, or fully speculative, one (valuation function $\mathcal{E}_{max}$).

The definition of the valuation functions $\mathcal{E}_{min}$ and $\mathcal{E}_{max}$ coincides for the pure functional syntactical constructors. Following Josephs' semantics [Jos89], the valuation function $\mathcal{E}$ is defined in terms of the auxiliary functions force and $\mathcal{A}$ (Fig. 2), which forces the evaluation of the variables in the environment, and applies functions to their arguments, respectively.

$$\mathcal{E}[\![x]\!]p\,\kappa = \mathsf{force}\ (p, x)\kappa$$

$$\mathcal{E}[\![\backslash x.e]\!]p\,\kappa = \kappa(\lambda x.e)$$

$$\mathcal{E}[\![e_1\ e_2]\!]p\,\kappa = \mathcal{E}[\![e_1]\!]p\,\kappa' \ \texttt{where}\ \kappa' = \mathcal{A}[\![e_2]\!]p\,\kappa$$

The evaluation of a variable implies a demand on the evaluation of the value associated to this variable. This is carried out by using the auxiliary function force (Fig. 2):

- When the value is already an expressed value, force just applies the corresponding continuation expression to this value.
- In the case of a closure, the continuation expression has to be modified to reflect, firstly, how the evaluation of the closure would modify the system, and, secondly, the updating of the variable.
- In the case of a communication, the closure must be evaluated in the adequate process, i. e. the producer. Once the communication value has been obtained:
  - Not only the variable, but also the channel have to be updated to the value (note that the consumer of the channel must be the initial process p).
  - Moreover, as a communication is taking place, the part of the environment corresponding to the free variables in the communicated value has to be copied from the producer to the consumer, using for this purpose the auxiliary function `nec` (Fig. 4) which will be explained in detail in Sec. 3.3.
- Finally, a self-reference (not_ready) or an undefined variable will produce an error, being called the continuation *wrong*.

The evaluation of a lambda-abstraction is intimately related with functional application. Because of the laziness of the language, the evaluation of the argument of an application has to be delayed until it is demanded. The auxiliary function $\mathcal{A}$ (Fig. 2) creates in the environment a new binding corresponding to the argument closure (which, thanks to the creation of a new identifier y, will only be evaluated once and shared by all the demands) and builds the continuation expression in case this argument is demanded. This continuation will have as argument the expressed value corresponding to the lambda-abstraction, i.e., a semantic function from identifiers to closures. In this case the continuation takes account of all the transformations that have to be delayed until the associated demand takes place.

$$
\begin{array}{l}
\mathsf{force} :: (\mathbf{IdProc} \times \mathbf{Ide}) \to \mathbf{ECont} \to \mathbf{Cont} \\
\mathsf{force}\ (p,x)\,\kappa = \lambda(\rho,\delta).\mathtt{case}\ \rho(p,x)\ \mathtt{of} \\
\qquad\qquad \varepsilon \longrightarrow \kappa\,\varepsilon\,(\rho,\delta) \\
\qquad\qquad \nu \longrightarrow \mathcal{E}[\![\nu]\!]p\,\kappa'\,(\rho',\delta) \\
\qquad\qquad\qquad \mathtt{where}\ \kappa' = \lambda\varepsilon.\lambda\Sigma.\kappa\,\varepsilon\,(\Sigma \oplus (\{(p,x) \mapsto \varepsilon\},\emptyset)) \\
\qquad\qquad\qquad\qquad\quad \rho' = \rho \oplus \{(p,x) \mapsto \mathsf{not\_ready}\} \\
\qquad\qquad (ch,\nu) \longrightarrow \mathcal{E}[\![\nu]\!]q\,\kappa'\,(\rho',\delta) \\
\qquad\qquad\qquad \mathtt{where}\ \kappa' = \lambda\varepsilon.\lambda\Sigma.\kappa\,\varepsilon\,(\Sigma \oplus (\{(p,x) \mapsto \varepsilon\},\{ch \mapsto (q,\varepsilon,p)\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \oplus(\mathtt{nec}\ [\![\varepsilon]\!]\ q\,x\,\kappa\,(\rho,\delta)\,p)) \\
\qquad\qquad\qquad\qquad\quad \rho' = \rho \oplus \{(p,x) \mapsto \mathsf{not\_ready}\} \\
\qquad\qquad\qquad\qquad\quad (q,\mathsf{unsent},p) = \delta(ch) \\
\qquad\qquad \mathtt{otherwise} \longrightarrow \mathit{wrong}\ (\rho,\delta) \\
\qquad\qquad \mathtt{endcase}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A} :: \mathbf{Exp} \to \mathbf{IdProc} \to \mathbf{ECont} \to \mathbf{ECont} \\
\mathcal{A}[\![e]\!]p\,\kappa = \lambda\varepsilon.\lambda(\rho,\delta).\,\mathcal{E}[\![\varepsilon\ y]\!]p\,\kappa\,(\rho',\delta) \\
\qquad \mathtt{where}\ y = \mathit{newIde}\,\rho\,p \\
\qquad\qquad\quad \rho' = \rho \oplus \{(p,y) \mapsto e\}
\end{array}
$$

Figure2. Auxiliary functions force and $\mathcal{A}$.

3.2 (Eager) Process creation

As we have explained before, process creation is not delayed until the value, which the new process has to produce, is demanded. By contrast, whenever an expression of process creation is detected, a new process is created in the system.

The differences between the maximal and the minimal semantics come off with the creation of new processes because, in the first case, the new process immediately evaluates its output, while this is only done under demand in the second case.

This is clearly expressed by the corresponding definitions of $\mathcal{E}_{min}$ and $\mathcal{E}_{max}$ for process creation, where $\mathcal{C}$ is an auxiliary function contained in Fig. 3:

$$
\begin{array}{l}
\mathcal{E}_{min}[\![e_1 \# e_2]\!]p\,\kappa = \lambda\Sigma.\mathcal{E}_{min}[\![x^{out}]\!]q\ \kappa\ \Sigma'' \\
\qquad \mathtt{where}\ (\Sigma',q,x^{in}) = \mathcal{C}\,(e_1,e_2)\,(p,x^{out})\ \Sigma\ \Sigma\ \kappa \\
\qquad\qquad\quad x^{out} = \mathit{newIde}\,\rho\,p \\
\qquad\qquad\quad (\rho,\delta) = \Sigma \\
\qquad\qquad\quad \Sigma'' = \Sigma'
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}_{max}[\![e_1 \# e_2]\!]p\,\kappa = \lambda\Sigma.\mathcal{E}_{max}[\![x^{out}]\!]q\ \kappa\ \Sigma'' \\
\qquad \mathtt{where}\ (\Sigma',q,x^{in}) = \mathcal{C}\,(e_1,e_2)\,(p,x^{out})\ \Sigma\ \Sigma\ \kappa \\
\qquad\qquad\quad x^{out} = \mathit{newIde}\,\rho\,p \\
\qquad\qquad\quad (\rho,\delta) = \Sigma \\
\qquad\qquad\quad \Sigma'' = \mathsf{force}\ (q,x^{in})\,\kappa\,(\mathsf{force}\ (p,x^{out})\,\kappa\,\Sigma')
\end{array}
$$

The minimal semantics ($\mathcal{E}_{min}$) just creates the new process and establishes the new channels, whereas the maximal semantics ($\mathcal{E}_{max}$) also forces the evaluation of both the output of the child process and the new output of the parent.

The actual creation of a process is carried out by using the auxiliary function $\mathcal{C}$ (Fig. 3), which gives back the extended environment and set of channels (that is, the modified system), the identifier of the new process, and the identifier associated to the communication from the parent to the child. When establishing a new communication, the new channel is initially set to unsent, while the corresponding closure is associated to a fresh variable in the consumer environment. The superscripts *in* and *out* indicate the direction of the communications, taking the point of view of the new process: *in* from parent to child and *out* from child to parent.

$$\mathcal{C} :: (\mathbf{Exp} \times \mathbf{Exp}) \to (\mathbf{IdProc} \times \mathbf{Ide}) \to \mathbf{Sys} \to \mathbf{Sys} \to \mathbf{ECont} \to (\mathbf{Sys} \times \mathbf{IdProc} \times \mathbf{Ide})$$

$$\begin{array}{l}
\mathcal{C}\,(e_1, e_2)\,(p, x)\,\Sigma_1\,\Sigma_2\,\kappa = (\Sigma'', q, x^{in}) \\
\quad \texttt{where}\ \Sigma'' = \Sigma' \oplus (\{(p, x) \mapsto (ch^{out}, e_1\ x^{in}), (q, x^{in}) \mapsto (ch^{in}, e_2)\}, \\
\qquad\qquad\qquad \{ch^{out} \mapsto (q, \mathsf{unsent}, p),\ ch^{in} \mapsto (p, \mathsf{unsent}, q)\}) \\
\qquad\quad q = newIdProc\ \Sigma_1 \\
\qquad\quad \Sigma' = (\rho', \delta') = \mathcal{N}[\![e_1]\!]p\ \kappa\ \Sigma_2\ q \\
\qquad\quad x^{in} = newIde\ \rho'\ q \\
\qquad\quad \{ch^{in}, ch^{out}\} = twoNewIdChans\ \delta'
\end{array}$$

Figure3. Auxiliary function $\mathcal{C}$ for process creation.

Note how the identifier x^{out}, representing the child output in the parent, is already passed as an argument to $\mathcal{C}$ instead of coming out as a result (like the one corresponding to the input from the parent in the child). Note also that the initial system Σ is passed twice when calling $\mathcal{C}$. In fact, the definition of $\mathcal{C}$ is a bit more complicated than needed in this particular case of process creation expressions, because it will be also used later when dealing with process creation at local declarations (see Sec.3.4).

3.3 Extending the environment

Process creation (and communication) involve copying some bindings from the parent to the child (from the producer to the consumer). The auxiliary functions $\mathcal{N}$ and `nec` (Fig. 4) are used to obtain the part of the environment which will be needed to evaluate the body of the newly created process (the communicated whnf). While $\mathcal{N}$ applies over expressions, `nec` is used to detect the dependencies inside values.

When $\mathcal{N}$ is applied to an expression, if it is a variable and there is no definition associated to that variable in the source process p, function $\mathcal{N}$ just returns the

$$\mathcal{N} :: \mathbf{Exp} \to \mathbf{IdProc} \to \mathbf{ECont} \to \mathbf{Sys} \to \mathbf{IdProc} \to \mathbf{Sys}$$

$$\begin{array}{l}
\mathcal{N}[\![x]\!]\, p\, \kappa\, \Sigma\, q = \texttt{case}\ \rho(p,x)\ \texttt{of} \\
\qquad\qquad \omega \longrightarrow \texttt{nec}\ \omega\, p\, x\, \kappa\, \Sigma\, q \\
\qquad\qquad \mathsf{undefined} \longrightarrow \Sigma \\
\qquad\qquad \texttt{endcase} \\
\qquad \texttt{where}\ (\rho,\delta) = \Sigma \\
\\
\mathcal{N}[\![\backslash x.e]\!]\, p\, \kappa\, \Sigma\, q = \mathcal{N}[\![e]\!]\, p\, \kappa\, \Sigma\, q \\
\mathcal{N}[\![e_1\ e_2]\!]\, p\, \kappa\, \Sigma\, q = \mathcal{N}[\![e_2]\!]\, p\, \kappa\, (\mathcal{N}[\![e_1]\!]\, p\, \kappa\, \Sigma\, q)\, q \\
\mathcal{N}[\![e_1 \texttt{\#} e_2]\!]\, p\, \kappa\, \Sigma\, q = \mathcal{N}[\![e_2]\!]\, p\, \kappa\, (\mathcal{N}[\![e_1]\!]\, p\, \kappa\, \Sigma\, q)\, q \\
\mathcal{N}[\![\texttt{let}\ \{x_i = e_i\}_n\ \texttt{in}\ e]\!]\, p\, \kappa\, \Sigma\, q = \Sigma_n \\
\qquad \text{where}\ \ \Sigma_0 = \mathcal{N}[\![e]\!]\, p\, \kappa\, \Sigma\, q \\
\qquad\qquad\ \ \Sigma_j = \mathcal{N}[\![e_j]\!]\, p\, \kappa\, \Sigma_{(j-1)}\, q \quad (1 \le j \le n)
\end{array}$$

$$\texttt{nec} :: \mathbf{Val} \to \mathbf{IdProc} \to \mathbf{Ide} \to \mathbf{ECont} \to \mathbf{Sys} \to \mathbf{IdProc} \to \mathbf{Sys}$$

$$\begin{array}{l}
\texttt{nec}\ \omega\, p\, x\, \kappa\, \Sigma\, q = \texttt{case}\ \omega\ \texttt{of} \\
\qquad\qquad \varepsilon \longrightarrow \Sigma \oplus (\{(q,x) \mapsto \omega\}, \emptyset) \oplus (\mathcal{N}[\![\varepsilon y]\!] p\, \kappa\, \Sigma\, q) \\
\qquad\qquad\qquad\qquad\qquad \texttt{where}\ y = \mathit{newIde}\ \rho\, p \\
\qquad\qquad\qquad\qquad\qquad\qquad (\rho,\delta) = \Sigma \\
\qquad\qquad \nu \longrightarrow \Sigma \oplus (\{(q,x) \mapsto \nu\}, \emptyset) \oplus (\mathcal{N}[\![\nu]\!] p\, \kappa\, \Sigma\, q) \\
\qquad\qquad (ch,\nu) \longrightarrow \texttt{nec}\ \omega'\, p\, x\, \kappa'\, \Sigma'\, q \\
\qquad\qquad\qquad\qquad\qquad \texttt{where}\ (\rho',\delta') = \Sigma' = \mathsf{force}\ \ (p,x)\, \kappa\, \Sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad \omega' = \rho'(p,x) \\
\qquad\qquad\qquad\qquad\qquad\qquad \kappa' = \lambda\varepsilon.\lambda\Sigma.\kappa\, \varepsilon\, (\Sigma \oplus \Sigma') \\
\qquad\qquad \mathsf{not_ready} \longrightarrow \Sigma \oplus (\{(q,x) \mapsto \omega\}, \emptyset) \\
\qquad\qquad \texttt{endcase}
\end{array}$$

Figure4. Auxiliary functions $\mathcal{N}$ and `nec`.

same system Σ. But when there is a value associated to the variable, function `nec` is called, which distinguishes four cases:

Expressed value: The value is a lambda abstraction and the possible dependencies are located in the corresponding closure. To reach this expression we provide, as argument, a fresh undefined variable y. In this way, the free variables of the closure can be located, while y does not extend the environment because it is undefined. So, the (initial) variable and the recollected bindings, which can be necessary to use the value, are added to the environment for the target process q.

Closure: When evaluating the variable, the process will need what is going to be demanded during the evaluation of the closure. Therefore, $\mathcal{N}$ is applied to the closure itself.

Communication: As only expressed values can be communicated through channels, the evaluation of the closure associated to the channel is forced, mod-

ifying, as a consequence, the system, in particular the channel set (this is the reason for $\mathcal{N}$ and `nec` returning **Sys**). Thereafter, `nec` is applied to the obtained value.

Evaluating: The copy of bindings cannot be fulfilled in this moment (not_ready).

3.4 Recursive local declarations

Valuation functions $\mathcal{E}_{min}$ and $\mathcal{E}_{max}$ also differ for local declarations, due to the possibility of having declarations like $x = e_1 \# e_2$, which would imply the immediate creation of a new process and, in the case of maximal semantics, forcing the evaluation of the new outputs. The corresponding definitions are as follows:

$$\mathcal{E}_{min}[\![\texttt{let } \{x_i = e_i\}_n \texttt{ in } e]\!]p\,\kappa = \lambda\Sigma.\mathcal{E}_{min}[\![e]\!]p\,\kappa\,\Sigma''$$

$$\begin{aligned}\texttt{where}\ \ & \Sigma_0 = \Sigma \\ & (\Sigma_i, q_i, x_i^{in}) = \mathcal{D}[\![e_i]\!]\,(p, x_i)\,\Sigma_{i-1}\,\Sigma'\,\kappa \quad (1 \le i \le n) \\ & \Sigma' = \bigoplus_{i=1}^{n} \Sigma_i \\ & \Sigma'' = \Sigma'\end{aligned}$$

$$\mathcal{E}_{max}[\![\texttt{let } \{x_i = e_i\}_n \texttt{ in } e]\!]p\,\kappa = \lambda\Sigma.\mathcal{E}_{max}[\![e]\!]p\,\kappa\,\Sigma''$$

$$\begin{aligned}\texttt{where}\ \ & \Sigma_0 = \Sigma \\ & ((\Sigma_i, q_i, x_i^{in}) = \mathcal{D}[\![e_i]\!]\,(p, x_i)\,\Sigma_{i-1}\,\Sigma'\,\kappa \quad (1 \le i \le n) \\ & \Sigma' = \bigoplus_{i=1}^{n} \Sigma_i \\ & \Sigma'_0 = \Sigma' \\ & \Sigma'_i = \begin{cases} \mathsf{force}\ (q_i\ , x_i^{in})\,\kappa\,(\mathsf{force}\ (p, x_i)\,\kappa\,\Sigma'_{i-1}) & \text{if } e_i \equiv e'_i \# e''_i \\ \Sigma'_{i-1} & \text{otherwise} \end{cases} \\ & \quad (1 \le i \le n) \\ & \Sigma'' = \Sigma'_n\end{aligned}$$

A new auxiliary function $\mathcal{D}$ (see Fig. 5) is introduced for dealing with each individual declaration: When the expression bound to the variable is not a process creation, then the environment is extended with the binding of the variable to the corresponding closure. On the other hand, when the expression is a process creation, the function $\mathcal{C}$ (Fig. 3), defined in Sec. 3.2 for creating a process, is used.

Due to the recursive nature of the `let` declaration, function $\mathcal{D}$ needs the actual system (Σ_{i-1}) obtained from the previous individual declaration, as well as the final system (Σ') corresponding to the complete local declaration. The former will be used to obtain fresh process identifiers, while the latter represents the system under construction and, consequently, the final system.

Example 2. The following example illustrates the difference between $\mathcal{E}_{min}$ and $\mathcal{E}_{max}$.

Let $P \equiv \texttt{let } x = y \# \backslash t.t,\ y = \backslash z.z \texttt{ in } y$, whose evaluation generates a speculative process.

$$\mathcal{E}[\![P]\!]p\,\kappa = \lambda\Sigma.\mathcal{E}[\![y]\!]p\,\kappa\,\Sigma'$$

$$\mathcal{D} :: \mathbf{Exp} \to (\mathbf{IdProc} \times \mathbf{Ide}) \to \mathbf{Sys} \to \mathbf{Sys} \to \mathbf{ECont} \to (\mathbf{Sys} \times \mathbf{IdProc} \times \mathbf{Ide})$$

$$\mathcal{D}[\![e_1 \# e_2]\!]\,(p,x)\,\Sigma_1\,\Sigma_2\,\kappa = \mathcal{C}(e_1,e_2)\,(p,x)\,\Sigma_1\,\Sigma_2\,\kappa$$

$$\mathcal{D}[\![e]\!]\,(p,x)\,(\rho_1,\delta_1)\,(\rho_2,\delta_2)\,\kappa = ((\rho_2',\delta_2),p,x) \qquad \text{if } e \neq e_1 \# e_2$$
$$\texttt{where } \rho_2' = \rho_2 \oplus \{(p,x) \mapsto e\}$$

Figure5. Auxiliary function $\mathcal{D}$ for a local declaration.

The minimal and the maximal semantics differ in the construction of Σ'. In the minimal, the first phase (i.e. the application of $\mathcal{D}$ to the two local expressions) consists of:

1. Creation of the process q associated to x using the function $\mathcal{C}$.
2. Incorporation of the closure associated to y to the environment.
3. Copy of the closure associated to y in the process q.

Obtaining:

$$\begin{aligned}\Sigma'_{min} = \Sigma \oplus (\{&(p,x) \mapsto (ch^{out}, y\,x^{in}), (q,x^{in}) \mapsto (ch^{in}, \backslash t.t),\\ &(p,y) \mapsto \backslash z.z, (q,y) \mapsto \backslash z.z\},\\ &\{ch^{out} \mapsto (q, \mathsf{unsent}, p), ch^{in} \mapsto (p, \mathsf{unsent}, q)\})\end{aligned}$$

Whereas in the maximal semantics a further step is needed:

4. Use of force to evaluate the channels between p and q.

Resulting:

$$\begin{aligned}\Sigma'_{max} = \Sigma \oplus (\{&(p,x) \mapsto \lambda t.t, (q,x^{in}) \mapsto \lambda t.t, (p,y) \mapsto \backslash z.z, (q,y) \mapsto \lambda z.z\},\\ &\{ch^{out} \mapsto (q, \lambda t.t, p), ch^{in} \mapsto (p, \lambda t.t, q)\})\end{aligned}$$

In both cases the evaluation continues by:

i. The evaluation of y in p, i.e. transforming the closure $\backslash z.z$ into the expressed value $\lambda z.z$.
ii. (p,y) is bound to the expressed value $\lambda z.z$.

Therefore, departing from an empty initial system, we obtain:

$$\begin{aligned}\mathcal{E}_{min}[\![P]\!]\,p\,\kappa\,(\emptyset,\emptyset) = (\{&(p,x) \mapsto (ch^{out}, y\,x^{in}), (q,x^{in}) \mapsto (ch^{in}, \backslash t.t),\\ &(p,y) \mapsto \lambda z.z, (q,y) \mapsto \backslash z.z\},\\ &\{ch^{out} \mapsto (q, \mathsf{unsent}, p), ch^{in} \mapsto (p, \mathsf{unsent}, q)\})\end{aligned}$$

and

$$\begin{aligned}\mathcal{E}_{max}[\![P]\!]\,p\,\kappa\,(\emptyset,\emptyset) = (\{&(p,x) \mapsto \lambda t.t, (q,x^{in}) \mapsto \lambda t.t, (p,y) \mapsto \lambda z.z, (q,y) \mapsto \lambda z.z\},\\ &\{ch^{out} \mapsto (q, \lambda t.t, p), ch^{in} \mapsto (p, \lambda t.t, q)\})\end{aligned}$$

□

So that $\mathcal{E}_{min}[\![P]\!]\, p\, \kappa \neq \mathcal{E}_{max}[\![P]\!]\, p\, \kappa$.

Example 3. When no speculation is generated $\mathcal{E}_{min}$ and $\mathcal{E}_{max}$ are equivalent.

Let us now consider $Q \equiv \texttt{let}\ x = y\#\backslash t.t,\ \ y = \backslash z.z\ \texttt{in}\ x$. So that

$$\mathcal{E}[\![Q]\!] p\, \kappa = \lambda \Sigma. \mathcal{E}[\![x]\!] p\, \kappa\, \Sigma'$$

In this case, Σ' is built exactly in the same way as for P in Ex. 2, but the evaluation of x has to be carried out afterwards. In the minimal semantics this evaluation implies the following:

1. Evaluation of x requires the evaluation of the output of q.
2. To evaluate its output, q demands the input from p.
3. The application $y\, x^{in}$ is evaluated and associated to (p, x).

By contrast, in the maximal semantics these steps were already accomplished in the construction of Σ'. In the end, both semantics describe the same behaviour.

$$\begin{aligned}\Sigma' = (&\{(p,x) \mapsto \lambda t.t, (q, x^{in}) \mapsto \lambda t.t, (p,y) \mapsto \backslash z.z, (q,y) \mapsto \lambda z.z\},\\ &\{ch^{out} \mapsto (q, \lambda t.t, p), ch^{in} \mapsto (p, \lambda t.t, q)\})\end{aligned}$$

and therefore we have $\mathcal{E}_{min}[\![Q]\!]\, p\, \kappa = \mathcal{E}_{max}[\![Q]\!]\, p\, \kappa$.

Note that for P in Ex. 2 process q is speculative whereas for Q it is not.

□

The previous examples have shown how in the absence of speculative computations, $\mathcal{E}_{min}$ and $\mathcal{E}_{max}$ generate the same denotation. However, when at least one of the created processes is not demanded, both semantics are different for the same program. Note that as a consequence of the copy of bindings when q is created, y is demanded to both p and q, thus generating duplicated work.

4 Conclusions and future work

We have developed a formal model for a lazy λ-calculus suitable for describing distributed processes. The denotation of a program will be the process system obtained after its evaluation. The model is effective for investigating the interplay between laziness and eagerness in the calculus, and for measuring the degree of speculative parallelism and the work duplication.

In [HO00] an operational semantics for that same calculus is given. That work is based on the operational semantics for parallel lazy evaluation presented in [BFKT00], which is, in turn, based on Launchbury's *natural* semantics [Lau93], where closures are modelled as name/expression bindings which are collected in a heap representing the program space. The denotational semantics that we have defined here is closely related with that operational semantics. All the

computations which can be obtained with the operational rules, can be placed between the two denotations defined in the present paper.

Actually, the calculus presented here is a simplification of the programming language Eden [BLOP96,BLOP97], which extends the lazy functional language Haskell [PH99] with a set of *coordination* features, aimed to express parallel and concurrent algorithms. These coordination features are based on two main concepts: *explicit management of processes* and *implicit process communication*. Like the simple calculus presented here, Eden combines lazy and eager evaluation, being lazy in the pure functional part of the language and eager in the case of process creation and communication value production. Using the semantic theory developed in this paper, we can associate for each program two denotations: a *minimal* one representing an "almost" completely lazy evaluation (processes are still eagerly created but their bodies are not evaluated until demanded) and a *maximal* one where laziness is restricted to functional application, so that the set of all the possible computations for a program will range between these minimal and maximal denotations. It is our aim to use this denotational semantics for comparing Eden programs, i.e. determining behavioural equivalence of process systems, supporting process refinement, and proving the correctness of program transformations like those that have been developed for that language in [PPRS00].

Other characteristics of Eden that we are currently incorporating in our calculus are streams (for communicating not only a single value but a list of them), channel tuples (to define processes with several inputs and outputs), and dynamic reply channels (to make more flexible the actual rigid hierarchical process structure). Eden also includes non-deterministic processes, that have not been considered in this presentation. In the presence of non-determinism, we would have to change our model in order to deal with sets of possible value results. That extended model will be used to prove the correctness of the program analysis in [PS01]. These extensions are also being incorporated into the operational model (see [HO01]).

Finally, this denotational semantics can be adapted to other (lazy) parallel/concurrent functional languages. In particular, it will be easy to obtain a denotational model for GpH [THM+96]. In fact, the basis of the related operational semantics mentioned above is the model for GpH in [BFKT00].

Acknowledgements
We would like to thank Hans-Wolfgang Loidl, Fernando Rubio and Phil Trinder for their comments on earlier versions of this semantics.

References

[Abr90] S. Abramsky. *Research Topics in Functional Programming*, chapter 4: The lazy lambda calculus, pages 65–117. Ed. D. A. Turner. Addison Wesley, 1990.

[BFKT00] C. Baker-Finch, D. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *ICFP'00*, Montreal, Canada, September 2000.

[BLOP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report 96/10, Reihe Informatik, FB Mathematik, Philipps-Universität Marburg, Germany, URL http://www.mathematik.uni-marburg.de/inf/eden/index.html, 1996.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *Workshop on High-level Parallel Programming Models, HIPS'97. In conjuntion with the IEEE International Parallel Processing Symposium, IPPS'97*, pages 120–124. IEEE Computer Science Press, 1997.

[DB97] M. Debbabi and D. Bolignano. *ML with Concurrency: Design, Analysis, Implementation, and Application*, chapter 6: A semantic theory for ML higher-order concurrency primitives, pages 145–184. Monographs in Computer Science. Ed. Flemming Nielson. Springer-Verlag, 1997.

[FH99] W. Ferreira and M. Hennessy. A behavioural theory of first-order CML. *Theoretical Computer Science*, 216:55–107, 1999.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[HM99] K. Hammond and G. Michaelson (editors). *Research Directions in Parallel Functional Programming.* Springer-Verlag, 1999.

[HO00] M. Hidalgo-Herrero and Y. Ortega-Mallén. A distributed operational semantics for a parallel functional language. In *Trends in Functional Programming (Selected papers of the 2nd Scottish Functional Programming Workshop)*, volume 2, pages 89–102. Intellect, 2000.

[HO01] M. Hidalgo-Herrero and Y. Ortega-Mallén. A distributed operational semantics for Eden. In *IX Jornadas de Concurrencia*, pages 129–144. Universitat Ramon Llull, 2001.

[Jos86] M. B. Josephs. *Functional programming with side-effects.* PhD thesis, Oxford University, 1986.

[Jos89] M. B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *POPL'93*, Charleston, 1993.

[PH99] S. Peyton Jones and J. Hughes (editors). Report on the programming language Haskell 98. URL http://www.haskell.org, February 1999.

[PPRS00] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In *Trends in Functional Programming (Selected papers of the 2nd Scottish Functional Programming Workshop)*, volume 2, pages 13–26. Intellect, 2000.

[PS01] R. Peña and C. Segura. Non-determinism analysis in a parallel-functional language. In *Selected Papers of the 12th International Workshop on Implementation of Functional Languages, IFL'00*, pages 1–18. LNCS 2011, Springer-Verlag, 2001.

[Rep92] J. H. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University (Department of Computer Science), 1992.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[THM+96] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.

On the Convenience of Using Explicit Batching in Video–on–Demand*

Juan Segarra and Vicent Cholvi

{jflor,vcholvi}@lsi.uji.es
Departament de Llenguatges i Sistemes Informàtics,
Universitat Jaume I, 12071 Castelló, Spain

Abstract. One of the methods for taking advantage of multicast services is the use of batching. With this technic, several request of the same video are grouped and transmitted together, using only the bandwidth required for one transmission. This method is commonly used in transmission of streamed data. In this paper we analyze the system performance with explicit constant batching, and demonstrate that a system without explicit batching performs better in terms of delays. We also propose a dynamic batching policy which improves the system performance both in mean and in maximum serving times.

1 Introduction

With the advancement of broadband network technology Video–on–Demand (VoD) services are becoming commonplace, specially in local residential areas. A VoD system is typically implemented by a client–server architecture supported by certain transport networks.

Among the many issues that have attracted the attention of researchers, optimizing the network channel demand is considered a major one. In order to solve that problem, VoD systems usually serve multiple clients at the same time by means of artificially delaying several requests for the same video [5, 6, 11, 13] and serving them at the same time. This technic is known as *batching*.

However, these artificially introduced delays may not be always necessary or even convenient. For instance, in a system with low load, all video requests could be served immediately, and the use of batching would be clearly inefficient, since it only introduces an unnecessary delay. Moreover, the amount of bandwidth used in a high load period is much bigger than in low ones, and since these networks have to afford this requirement, in low load periods this resource is mostly underutilized despite of offering Internet connections or other similar services. Thus, bandwidth is not always a critical resource.

In this paper we focus on the convenience of using explicit batching. We will evaluate, by means of some simulations, how the use of batching improves (or reduces) the overall performance of the system described in Section 2. We

* This study is partially supported by the CICYT under grant TEL99-0582 and Bancaixa under grant P1-1B2000-12.

consider the *serving time* (the time between a video request and the beginning of its transmission) as the metric to evaluate the system performance.

First, we use a *Constant Explicit Batch–Time Policy* with different delays. Whereas this policy is the most widely used, we show that, in general, it provides improvements on bandwidth, but it does not provide any delay improvement against a system without explicit batching (but still with implicit batching). Then, we introduce a new batching policy which varies dynamically with the system load. Even though such a policy has a better performance in starting delays than previous ones, it only provides a low improvement when the system is highly saturated.

The rest of our paper is organized as follows. In Section 2 we describe our video–on–demand delivery system and the considered scenario. In Section 3 we show our comparison between constant batching policies and in Section 4 we present our approach and results with dynamic batching. Finally, our conclusions are presented in Section 5.

2 System

For our study, we use a quite simple network architecture (Fig. 1) consisting of a central server, storing all videos. Users are connected to that server by means of a common link which has a bandwidth of 1.5 Gbps downstream. A moderate bandwidth is also needed upstream in order to send user requests to the central server. We consider that this is a VoD dedicated network[1], therefore our goal is to obtain the best results using the available resources.

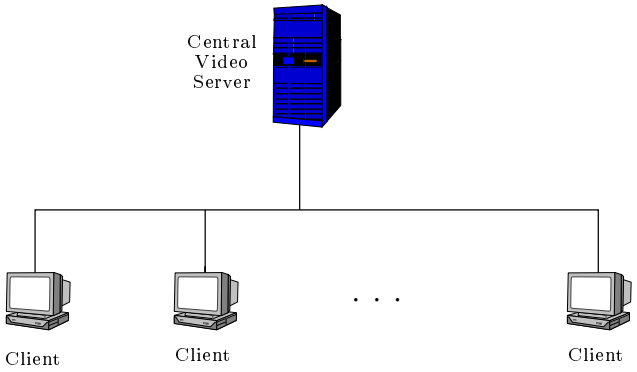


Fig. 1. Structure of our system.

We use a configuration consisting of 1 000 videos and 15 000 users. Each video information has been constructed making variations over the example presented

[1] Usually these networks offer more services, but dedicating a concrete amount of bandwidth to each service is also usual.

in [12], resulting in videos of a mean duration of about 85 minutes and a mean bandwidth requirements of about 1 850 Kbps in fragments ranging from 96 to 4 608 Kbps. With this configuration we ensure that the link will become *saturated* (i.e., it will reach an utilization of 100%) at some period of time, being non-saturated during the remaining time. That will allow us to analyze the system behavior in both situations.

On the other hand, transmitting real–time VBR flows is not a trivial matter because a different amount of bandwidth will be needed during the transmission [3, 8]. One of the methods to improve these transmissions is the *smoothing* [7, 9, 12] of streams before their transmission. This way peaks and rate variability requirements are minimized. We use an approach similar to [12], which develops a *transmission plan* consisting of time periods so that, in each period, the transmission may be performed by using a constant–bit–rate (CBR) network service. Thus, each video is characterized by a collection of tuples $\langle time, rate \rangle$, whose first parameter denotes a time period and whose second one is the CBR rate associated with such a time period.

In our approach to the video transmission problem we study the performance in a 24 hour period. Previous studies on video demand rates [2, 10] have determined the behavior of these rates in this period, and our simulations work with a rate distribution according to these studies. We also consider that a user makes a request only if it is not waiting nor receiving a video (i.e., nobody can receive more than one transmission at a time). In Fig. 2 we can see the statistical request rate and the request rate distribution obtained in a simulation having one request per day for each user. This is what we assume in our study (actual VoD trials show that the average number of user request per week is between 2 to 3).

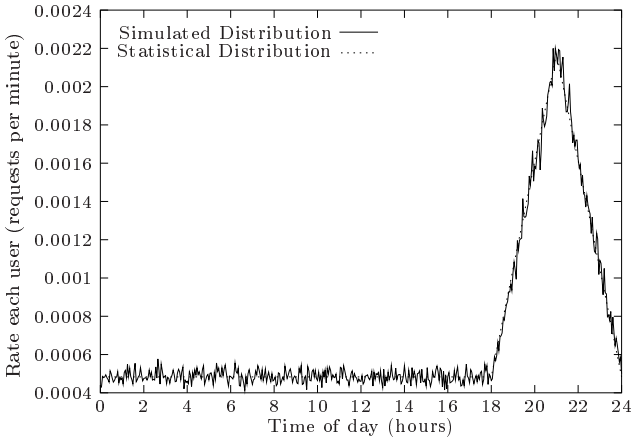


Fig. 2. Request rate distribution in a 24 hour period.

Furthermore, each video has a *popularity*, which represents the probability of being requested according to Zipf's law [4]. This distribution has been found to statistically fit video program popularities estimated through observations from video store statistics [1].

In order to guarantee that, once a video is accepted (maybe after some start–up delay), it will be delivered without interruption to the final user/s, it is necessary to use a *reservation algorithm* to manage video requests. It works as follows: when a video request is received, we test (for each video–part) if it is already planned for transmission at the same time. If so, that request is added to the multicast transmission of this video–part without any additional bandwidth requirements. Otherwise, we check if there is enough bandwidth during the slots of the transmission. In that case, bandwidth is reserved by adding the new video–part rate to the bandwidth used. See Fig. 3 for details.

Step #1: A user requests a video.
Step #2: If that video has been already requested and it has not started yet, the start time of the new request will be the same as the previous one. Otherwise, the new request will start after an explicit batch–time.
Step #3: The system calculates the exact time when each video–part has to be transmitted.
Step #4: For each video–part do
    If there is a reserve for this video–part at the same time, add our request as a multicast. Otherwise, reserve the bandwidth needed for this video–part during the required time slots.
Step #5: Accept video for transmission or go to Step #3 in the next time slot (in which case, all reserves and multicasts for this request performed in Step #4 are canceled).

Fig. 3. Algorithm for the acceptance of video requests.

As it can be seen, this reservation algorithm guarantees that all requests will be served, maybe after some start–up delay. Moreover, users can know the exact time when the transmissions will begin, so we assume that there are no cancellations.

An important feature of the reservation algorithm is the fact that, in saturation periods, all new requests of the same video will be accepted as multicast at the beginning of the transmission of that plan. So, even in the case where the there is not explicit batch–time, the reservation algorithm will still induce an *implicit batching* that, as we will show in the next sections, provides a very good system's performance.

3 Constant Batching Policies

Using a constant explicit batching scheme is currently the most used policy maybe because of its simplicity. Basically it consists of taking a constant batch–time delay throughout the whole system execution.

In this section, we analyze the performance of our system when taking explicit batch–time values of 0, 10, 20 and 30 minutes (despite we have tested more values, for clarity we only present results for these four values, since we found the rest follow the same pattern). Fig. 4 shows the starting delays in a 24 hour period.

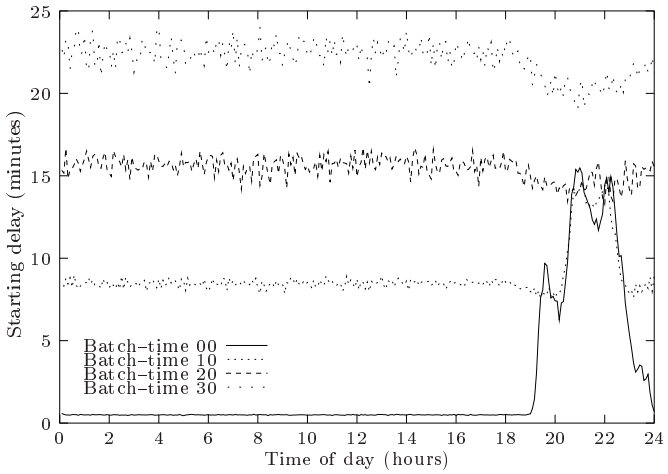


Fig. 4. Mean starting delays during a 24 hour period.

First of all, it can be readily seen that, when the request rate is low (from approximately 0h to 19h), the more we increase the batch–time the more the starting delay increases. On the other hand, Fig. 4 also shows that during the time period when the request rate is high (from approximately time 19h to 24h) starting delays are all following the high audience peak independently of the batch-time used. This behavior can be explained if we look at the reservation algorithm in Fig. 3. This algorithm tries to allocate transmissions as soon as possible after their batch–time. However, in the time interval where the system is saturated, the delay necessary to start transmitting is higher than the batching delay. Consequently, since the system gets saturated in a short time interval (a few minutes), at the very end, all policies operate in the same way most of this time period, confirming what has been observed in Fig. 4.

In Fig. 5 we can see that, in the low load period, bandwidth usage never gets saturated. Therefore, since there is enough bandwidth to serve immediately all requests, adding a batch–time will only increase their starting delays. In turn,

taking smaller batch–times will increase the used bandwidth. However, here we are assuming that the whole bandwidth is completely dedicated to our video–on–demand system. On the other hand, it can be seen that with very high explicit batch–times it is possible to avoid saturation. However, it also prevents from obtaining better starting delays.

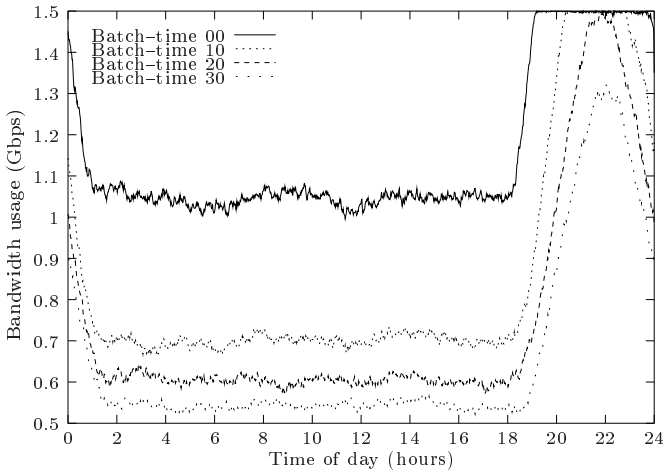


Fig. 5. Bandwidth usage during a 24 hour period.

In order to see how multicast usage varies in time, we include Fig. 6. In this figure it can be seen how, during the low load period, multicast capabilities are almost unused without explicit batching. On the other hand, during the high load peak, these capabilities are used quite similarly, which demonstrates that the reservation algorithm is working independently of the explicit batching. This shows that in saturation periods, explicit batch–times practically does not affect the behavior of the algorithm except when the explicit batch–time is higher than the required starting delay. High variability of values in this figure is produced by the batching effect.

4 Dynamic Batching

We have demonstrated, in the previous section, that a policy without explicit batching performs better in delay terms than with constant batch–times. In this section we present a dynamic batching approach to the problem. Such an approach consists of obtaining a batch–time that adjusts itself according to the current system load.

Our idea is the following. Think that the system receives a video request in the instant t, and the objective is to compensate the load we will have in the

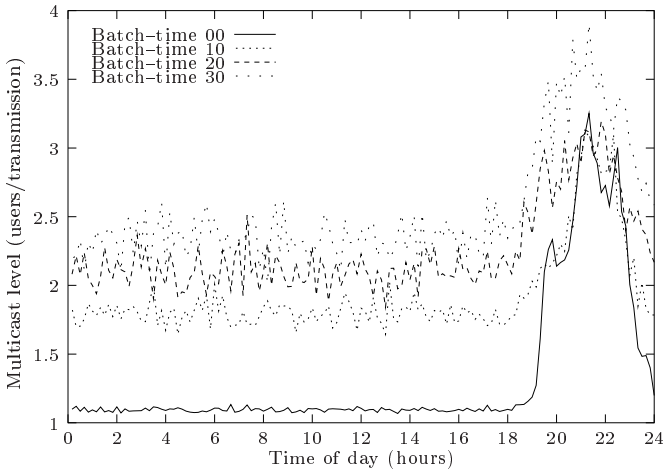


Fig. 6. Multicast level during a 24 hour period.

instant $t + \Delta t$. This compensation is done using batching over the requests in the current time t.

The batch–time we assign to a request on time t is defined as follows:

$$batch_t = load_{t+\Delta t}\, adjust$$

where $load_{t+\Delta t}$ represents the number of videos awaiting for transmission in the time slot $t+\Delta t$ and $adjust$ is a function that will be used to adjust the load with the batch–time. In order to make load independent of the system, $load_{t+\Delta t}$ has been normalized so that a value of 1 indicates the saturation point.

To define $adjust$, think in a system which is continuously receiving requests and immediately serving those requests at the same rate. This system has, in addition, a list of video requests waiting for being served. Therefore, when a request for a given video is received it may happen that that video is also in the list waiting for being served. Thus, both requests would be transmitted as a multicast and the additional load list would decrease in one video. If this procedure is used until all waiting videos in the list are transmitted, at that time we would have eliminated the additional load list. Let us call $EliminationTime$ the time where the additional load is eliminated.

$$EliminationTime = AdditionalLoad\, \frac{1}{ProbCoincidence}$$

where $AdditionalLoad$ is the additional load list (normalized as above) we have to eliminate, and $ProbCoincidence$ is the probability of having a coincidence when requesting a video for transmission, that is, the probability of the requested video being in the load list.

For our work, we choose $batch_t \equiv EliminationTime$. Since we have that $load_{t+\Delta t} \equiv AdditionalLoad$, then $adjust \equiv \frac{1}{ProbCoincidence}$. Therefore,

$$batch_t = \frac{load_{t+\Delta t}}{ProbCoincidence}$$

Because of the space limitation, here we do no provide a detailed description of $load_{t+\Delta t}$ and $ProbCoincidence$. However, they can be found in the Appendix.

As it can be seen in Fig. 7, our statistical load is a good approximation to the real one. Local peaks in the real system load are due to the system saturation and then a considerably reduction of this load with the transmission of many videos at the same time.

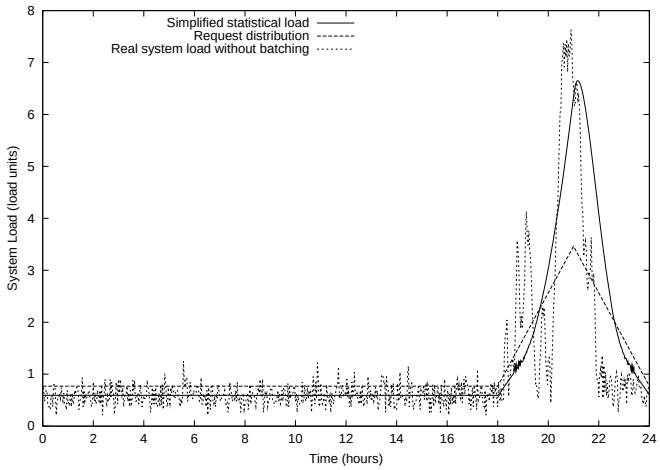


Fig. 7. Several representations of the system load: the real system load in a system without explicit batching, the statistical load and the request distribution as a reference.

Results

Now, we show a comparison between this policy and one without explicit batching (which has been shown in the previous section to provide better results than using any explicit constant batching policy).

Fig. 8 shows the starting delays both without explicit batching and with a dynamic batching policy. During the low load period, both policies perform very similarly and between 0 and 1 minute. When the high load period begins, the dynamic policy responds adding artificial delays for grouping requests. This action produces a sooner increase in starting delays but this prevents starting

delays from increasing so much as without explicit batch–time when these delays are higher. That is because the dynamic policy has a look ahead component, whereas the batch 0 policy has no information about the future system load.

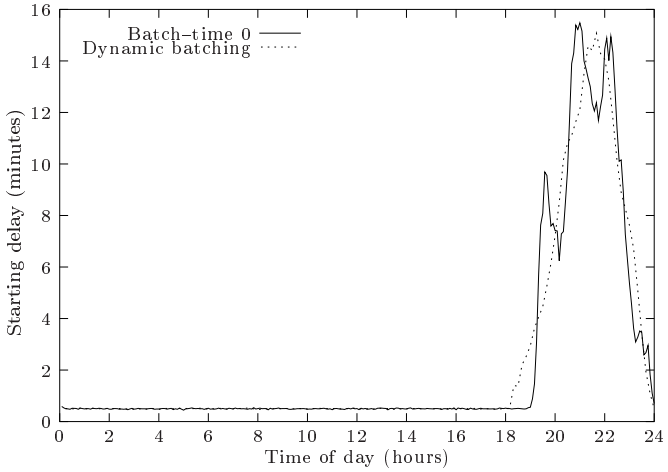


Fig. 8. Starting delays during a 24 hour period.

The mean starting delay during a 10 days period with the dynamic policy in this case is 4 min 21 sec, and its maximum delay has been 27 min 22 sec. These values are about a 0.4% and a 4.7% lower than without explicit batch–time.

Bandwidth usage with these configurations is presented in Fig. 9. Whereas the bandwidth usage is almost the same, with the dynamic policy the added batching produces a delay in saturation. The reason is the same as above, grouped requests are transmitted together in multicast transmissions and they use less bandwidth.

In Fig. 10 we can also see how multicast levels are very similar without explicit batch–time and with our dynamic batching policy. It is slightly over 1 user/transmission in low load period, and following a similar peak during the high request rate. As before, in this figure it can be seen how multicast transmissions begin before in the high rate period, with a better load adaptation when the load grows.

5 Conclusions

In this paper we have studied the effect of using batching in the transmission of video on demand. Instead of using cost functions, this study presents results based on the delays between video requests and the beginning of their transmission.

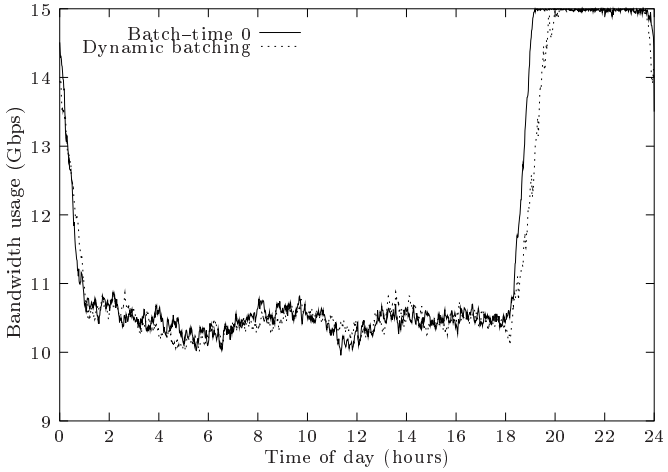


Fig. 9. Bandwidth usage during a 24 hour period.

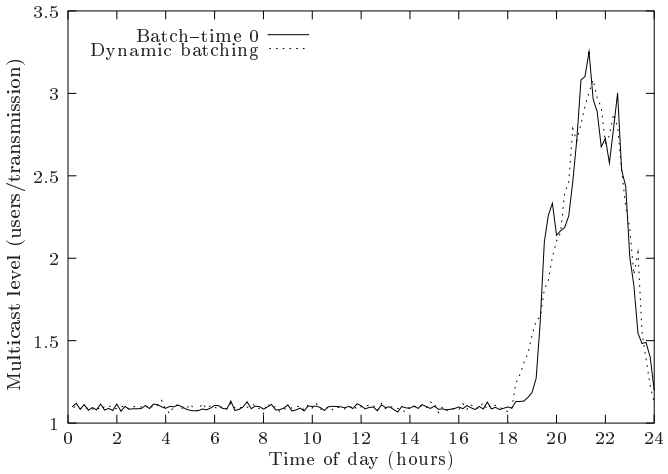


Fig. 10. Multicast level during a 24 hour period.

Firstly, we have demonstrated that, without using explicit batching techniques, the overall system performs better than using a constant explicit batch–time.

Furthermore, to improve the system performance, we have proposed a method for statistically calculate the system load, and use the resulting value to adjust the batch time dynamically. With this method we have reduced around 0.4% the mean starting delay and around 4.7% the maximum starting delay compared with a system without batching.

These results demonstrate that, unless a constant explicit batch–time is needed for some specific reason, it is better not using it. In case of needing a more optimized system, we also demonstrate that our dynamic batching method is better than using a batch–time 0 policy. However, the improvements are relatively small and require additional computation, so a policy without explicit batching could also be considered as a good option due to it performs well and it is the simplest batching policy. The reasons of the good performance are that without explicit batching there is no artificial delay introduced in the system, and this policy has a good adaptation to the current load, since it allocates the transmissions as soon as possible.

References

1. A. Dan and D. Sitaram and D. Shahabuddin. Dynamic Batching Policies for an On–Demand Video Server. *Multimedia Systems*, 4:112–121, 1996.
2. Bell Atlantic. Fact Sheet: Results of Bell Atlantic Video Services. Video–On–Demand Market Trial. Trial Results, 1996.
3. L. Berc, W. Fenner, R. Frederick, and S. McCanne. Rpt payload format for jpeg-compressed video. Request for Comments 2035, Network Working Group, October 1996.
4. Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the INFOCOM '99 conference*, March 1999.
5. Shueng-Han Gary Chan and Fouad Tobagi. Providing on-demand video services using request batching. *IEEE Int. Conf. Communications (ICC'98)*, pages 1716–1722, June 1998.
6. Asit Dan, Dinkar Sitaram, and Perwez Schahabuddin. Scheduling policies for an on-demand video server with batching. *ACM Multimedia*, pages 15–23, 1994.
7. W. Feng, F. Jahanian, and S. Sechrest. An optimal bandwidth allocation strategy for the delivery of compressed prerecorded video. *Multimedia Systems*, 5(5):297–309, 1997.
8. D. Gall. Mpeg: a video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.
9. Jeniffer Rexford and Don Towsley. Smoothing Variable–Bit–Rate Video in an Internetwork. *IEEE/ACM Transactions on Networking*, pages 202–215, April 1999.
10. PG de Haar et al. DIAMOND Project: Video–on–Demand System and Trials. Eur Trans Teleccommun (8)4: 337–244, 1997.
11. Juan Segarra and Vicent Cholvi. Distribution of video–on–demand in residential networks. 8^{th} *International Workshop on Interactive Distributed Multimedia Systems*, September 2001. Lecture Notes in Computer Science 2180.

12. Arun Solleti and Kenneth J. Christensen. Efficient transmission of stored video for improved management of network bandwidth. *International journal of network management*, 10:277–288, 2000.
13. Constantinos Vassilakis, Michael Paterakis, and Peter Triantafillou. Video placement and configuration of distributed video servers on cable TV networks. *Multimedia Systems*, 8:92–104, 2000.

Appendix: Description of Dynamic Batching Parameters

Definition 1 *We define the* real load *of a video–on–demand system in a time slot t as:*

$$rload_t = \frac{|(Awaiting_{t-1} \cup Requested_t) - Served_t|}{ServingCapacity}$$

where $Served_t$, $Requested_t$ and $Awaiting_t$ respectively denote the set of different videos served, requested and awaiting for transmission in the time slot t, and $ServingCapacity$ is the mean number of different videos served in a time slot using all bandwidth.

Note that $ServingCapacity$ is used to normalize the real load so that a system working at full capacity without being saturated will have a load value of 1.

Nevertheless, this definition is not usable in practice, because it needs both which videos are awaiting for transmission and which of them will be served in each time slot. Obviously, it is not known in advance which videos will be requested, nor which ones will serve the system since that depends on the batch–time assigned to each request.

Therefore, we need to obtain a (statistical) load value independent of the batching algorithm. All probability operations below are done using the Zipf's distribution, which offers probability values based on popularity.

Definition 2 *We define the* statistical load *of a video–on–demand system in a time slot t as:*

$$sload_t = max\left(0, sload_{t-1} + \frac{NewRequests_t - ServingCapacity}{ServingCapacity}\right)$$

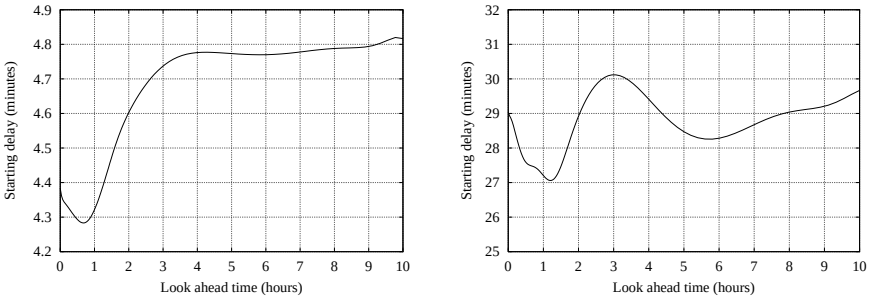
In this equation, we calculate the load in a time slot t by adding the load generated by the new requests to the load of the previous time slot and subtracting the amount of load transmitted, which is 1 because of the normalization[2].

The $NewRequests_t$ parameter is the number of new requests in the time slot t, and it is calculated as:

$$NewRequests_t = ProbNewReq(ReqInSlot(t), sload_{t-1}\, ServingCapacity)$$
$$ReqInSlot(t)$$

where $ReqInSlot(t)$ is the number of requests in the time slot t obtained statistically using the request distribution and $ProbNewReq(new, awaiting)$ is the probability of that *new* requests not being included in *awaiting* requests. Statistically $ProbNewReq(new, awaiting)$ is calculated as $P(Videos(new) \mid Videos(new) \nsubseteq Videos(awaiting))$, being $Videos(x)$ a set of x $VideoID$s obtained statistically from the repository using Zipf's law. Note that $Videos(x)$

[2] Actually, we do not know the value representing the transmitted videos, so we assume the system is serving at full capacity (1 because of the normalization) and then use the *max* operation to prevent the system from getting load values lower than 0.



(a) Mean starting delays (b) Max. starting delays

Fig. 11. Starting delays for different Δt.

must have equal or less (if some videos are requested several times) elements than x.

However, obtaining this statistical load value has a computation cost of $\mathrm{O}(MaxVideos^{new})$ in the step $ProbNewReq(new, awaiting)$, so an approximation is needed. We obtain this approximation supposing that the proportion of different requests in each time slot is a constant k, and the $sload_{t-1}$ videos awaiting for transmission are the most popular ones. The first simplification is reasonable when the number of requests in each time slot is not very different and in our case it is less than one order of magnitude. The second one is reasonable having in mind that Zipf's law popularity implies exponential probabilities of request, so the most popular videos are much more requested than the others. The usable $ProbNewReq(new, awaiting)$ definition would be the probability of not requesting the $awaiting$ most popular videos. Statistically: $k\,P(x \mid x > awaiting)$, where k is the proportion of distinct elements in a group of $VideoID$s requested in each time slot.

With these modifications, the statistical load in *Definition 2* is independent of the batching algorithm, so it can be used for our purpose.

The remaining definition is $ProbCoincidence$, but it is exactly the opposite of $ProbNewReq()$, so its calculation is immediate.

In order to set up the parameters we use in our algorithms, there are two parameters which can be obtained easily. $ServingCapacity$ is obtained dividing the link capacity by the product of the mean bandwidth usage of all videos each minute and the mean video duration. The resulting value was 9.42 videos. The parameter k is obtained using the mean value from a complete simulation. The resulting value was $k = 0.77$.

To obtain the best value of Δt, we have simulated our system with Δt ranging from 0 to 10 hours. Figures 11(a) and 11(b) show the obtained results for both the

mean and the maximum starting delays[3]. Since both figures have the minimum at approximately the same value, this means that a good Δt can improve both the mean and the maximum starting delays. Depending on the pursued objective this value can be adjusted closely, in our case we take a value of 60 minutes, which is between the best mean delay and the best maximum delay.

[3] These figures have been smoothed with a Bezier method in order to get a better view of the plot tendency.

A Bluetooth-based Location Network

Jaime García-Reinoso, Francisco J. González-Castaño, Pedro S. Rodríguez-Hernández, and José M. Pousada-Carballo

Departamento de Ingeniería Telemática, Universidad de Vigo.
{reinoso,javier,pedro,chema}@det.uvigo.es

Abstract. In this paper, we propose an *auxiliary location network*, called Bluetooth Location Network (BLN), for location-aware or context-driven mobile networks. BLN is composed of small, completely independent, Bluetooth wireless nodes, which establish an spontaneous network topology at system initialization. Its users carry either a Bluetooth-enabled handheld or a Bluetooth *badge*. BLN has self-diagnosis capability: in case of multiple node failures, BLN reconfigures itself spontaneously.

1 Introduction

1.1 Motivation

In this paper, we propose an auxiliary location network for location-aware or context-driven mobile networks. Some examples of context-driven network users are walking individuals that shop in a mall (m-commerce), people visiting a museum (e-museums) or professionals that interact with exhibition stands.

For instance, m-Commerce (mobile e-commerce, [1]) for cell phones or PDAs [2, 3] has a promising future. Datamonitor [4] has predicted that the US m-commerce market will grow 1,000 percent up to 1.2 billion US$ by 2005. In a typical m-commerce scenario, customers walk around a large commercial area or mall carrying wireless PDAs. A PDA client allows its user not only to purchase items, make reservations or request information, but also to receive (possibly context-driven) store coupons, advertisements, advice and guidance.

Another interesting application field is electronic guidance for museums or exhibitions. Visitors receive specific information associated to their current location. See [5] for a review of current initiatives.

In any of those scenarios, there may exist information servers that need to know user location in real-time, and send context-oriented information to user handhelds when necessary.

The auxiliary location network carries position information to *service servers*, without user participation. We are interested in a general-purpose RF base technology (without line-of-sight constraints), preferably available in commercial handhelds as an option. The auxiliary location network should admit alternative use as a data transmission network, if required (for example, as a security network when the target area is closed to the public, or as a spare network for emergencies).

As a particular case, we describe an auxiliary Bluetooth Location Network (BLN). BLN users carry either a Bluetooth-enabled handheld [6, 7] or a Bluetooth badge (in this case, BLN provides location services to any kind of user terminal). BLN is composed of small, completely independent Bluetooth nodes (no wires), which establish an spontaneous network topology at system initialization. BLN can coexist with Bluetooth devices that are not part of the location system, such as printers or headphones. BLN has self-diagnosis capability. Node failure does not compromise BLN operation (i.e. maintenance is not critical), although it may decrease location precision. Either at initialization, or in case of multiple node failures, BLN reconfigures itself spontaneously. As far as we know, operation survivability was not a goal in previous user-positioning research.

1.2 Background

- *Outdoor systems*: cell phone location services [8] and GPS [9]. These systems are limited to outdoor applications (specially GPS). In that case, they are quite effective, and possibly the best choice.
- *Indoor systems*: for indoor applications, there exist several alternatives:
  - *RF triangulation-oriented systems*: Pinpoint's 3D-iD system [10] is based on *cell controllers*, which interact with RF *tags*. The cell controller is linked to several antennas, that receive tag identifiers. User position can be determined by processing tag signals received by different antennas. From our point of view, it has two disadvantages: It needs coaxial cables to link cell controllers with antennas, and cannot be used as a data network for alternative purposes.
  - *Beacons*: HP's CoolTown [11] is based on IR beacons, that push position-dependent URLs into handheld IR ports. The system is user-dependent, since he must aim the infrared port to location beacons. It could be argued that automatic detection of location information (without user participation) may have severe consequences in terms of nuisance value. A good example happens when users are annoyed because the Web page they are viewing is suddenly supplanted by one advertising frozen peas from a grocery store nearby. Obviously, we must expect the system to be *reasonable*. Consider, for example, a museum, where updates are associated to new halls, once the user enters them, and imagine that the current page is reloaded with a tiny icon at its bottom meaning "do you want to update context information"? On the other hand, asking the user to locate IR beacons each time he enters a room full of visual distractions may be tiring, and signaling beacons with large red arrows unsightly. Possibly, depending on the specific application, user-dependent line-of-sight IR systems may be more advantageous than user-independent ones or viceversa, and both approaches deserve consideration. Other examples of IR systems are described in [12, 13].
  - *Client processing*: In these systems, clients calculate their position using beacon signals as references. For example, MIT Cricket [14] uses a combination of RF and ultrasonic inputs. The authors claim that, by freeing

system servers of location processing, their system is more scalable. On the other hand, in an user-independent location system, user handhelds should interact with Cricket non-standard hardware to extract position information, which should be sent to the servers via some data network. Such interaction is not described, and would increase handheld complexity to some unknown degree.

- *Pinging*: In these systems, the beacons transmit signals that trigger client pulse transmission. In AT&T Bat [15], the beacons are linked to DSP-controlled sensor arrays that receive ultrasonic pulses generated by the clients, determine user position, and transmit it to servers via a specialized data network. This approach is user-independent and the data network can be shared with other applications. However, Bat terminals are non-standard, and sensor arrays are considerably complex.

Note 1: As far as we know, none of the user-positioning systems above have considered operation survivability in case of failure as a research goal.
Note 2: It should be understood that the philosophy in section 1.1 is not necessarily the best one for *any* application. Each system, including ours, has clear advantages over the remaining ones, based on its *own* assumptions. For example, Bat sensor array complexity implies a high location precision. Cricket is extremely simple, and works well in case of user-dependent location services. A single 3D-iD cell controller may handle a medium-sized store. Finally, Cooltown may use IR ports present in most handhelds.

1.3 m-Mall

The m-Mall sistem [16] follows the philosophy in section 1.1 to some extent. M-Mall users carry a Bluetooth-enabled terminal, or any mobile terminal and a Bluetooth location badge. The badge interacts with a Bluetooth [17] location network, which provides servers with real-time user position. The servers may use this information to push URLs into user terminals via TCP/IP sockets, or to update WAP cards. Thus, no client action is required to generate context-driven updates.

The m-Mall architecture in [16] had fixed *routing tables*. Therefore, its configuration does not change automatically in case of failures and, consequently, its maintenance would require constant monitoring. In this paper, we generalize the m-Mall location network and define a new auxiliary Bluetooth Location Network (BLN) that fulfils the requirements in section 1.1. Like in m-Mall, the customer must access the service Web/WAP server from his handheld, and enter his badge number. By doing so, the Bluetooth address of the badge becomes *valid* from BLN's point of view (obviously, the location network must work even if invalid addresses are present, as we will see later). The servers associate the client's IP address or WAP session to its badge number, for all subsequent transactions.

The rest of this paper is organized as follows: the next section describes BLN protocols. Section 3 analyzes BLN survivability issues: reconfiguration, surviving detection capabilities and isolated operational nodes in case of failure. Finally, section 4 concludes.

2 BLN protocols

2.1 Bluetooth

Bluetooth [17] is a short-range wireless technology. The maximum separation between devices is 10 m [18]. It was originally intended for short-range applications such as wireless headphones and computer-to-peripheral communications.

A Bluetooth network is composed of *piconets*. A piconet is a spontaneous network whose configuration changes whenever a Bluetooth device enters or leaves its range. A piconet has one master and up to seven *active* connected slaves [17].

It is possible to create nets of piconets, or *scatternets*. In a scatternet, the master of a piconet can be a slave of another piconet. In this scheme, inter-piconet communication is allowed. Note that the position of a given slave can be determined with 10-m precision in a scatternet, in the worst case.

From our point of view, the main advantages of this technology are its low power consumption ($\sim$ 1 mA standby and $\sim$ 60 mA peak) [19], small modem size (1 cm^2) and low cost previsions (\$5-\$10) [20–23]. Note that these previsions would allow the installation of a large number of piconets in a target area.

Consequently, Bluetooth seems a good alternative to deploy an auxiliary location network. Next, we will describe BLN architecture. Nevertheless, it should be understood that the ideas proposed in this paper could be implemented on any adequate short-range wireless technology, such as SPIKE [24].

2.2 BLN configuration

The auxiliary BLN is composed of mobile badges and static Bluetooth units (located in the ceiling, for example). We will refer to the latter as *static nodes*. Static nodes (SNs) are arranged in a network that covers the whole target area. Hexagonal tiling is a typical solution in cellular network planning, which we have followed in this research (figure 1). Other arrangements such that any SN has at most seven closest neighbors located less than 10 m away could be used as well. For example, a mesh for 2D target areas, or a k-ary 3-cube [25] for 3D target areas.

Each cell in figure 1 has an area of 86.55 m^2. SN units scan their surroundings periodically, by means of Bluetooth inquiry calls [17]. All SNs are organized in a radial scatternet around a *master node*, SN0, connected to the servers (not shown). The remaining SNs are arranged in "circular" layers around SN0. The notation SNX-Y stands for the Bluetooth address of SN Y in layer X. In layer X, SNX-1 is placed right above SN0, and the remaining Y values are increased clockwise. Our example shows the six cells in the first layer, SN1-1 to SN1-6, and two cells in the second layer, SN2-3 and SN2-4. Each SN is a slave of all six surrounding neighbor SNs, and therefore all SNs have six slaves.

All SNs perform inquiry cycles periodically, to publish their existence. If SN a detects an inquiry from SN b, and b is not currently listed in a's routing table,

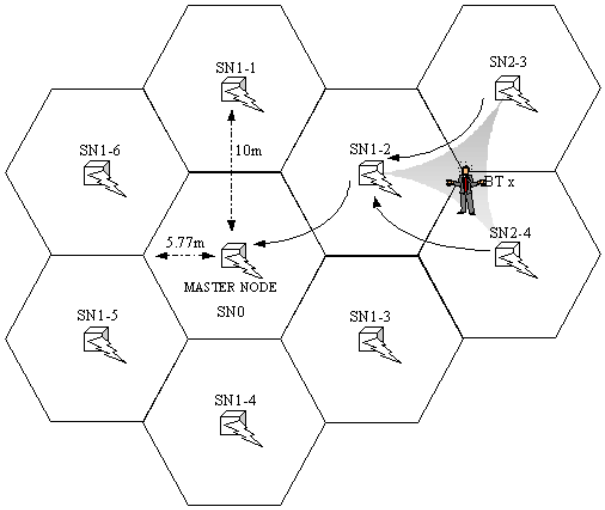


Fig. 1. Bluetooth Location Network

a must send its *minimum distance to the master node*[1] to *b*. All SN minimum distances are set to ∞ at power up, excepting the master node's, which is set to 0. Thus, the master node initiates the routing configuration process by sending 0-hop distance packets to its neighbors. Later, if a SN performing an inquiry cycle does not receive an answer from one of its neighbors that was previously listed in the routing table, it deletes the corresponding table entry. If, as a consequence, its minimum distance to the master node changes, it transmits a distance packet to all its slaves, indicating the change.

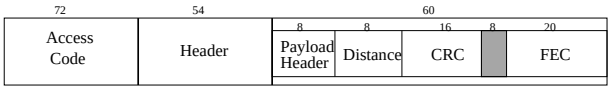


Fig. 2. Distance packet format

Whenever a SN receives a distance packet, it searches its routing table to check if the corresponding distance is lower than its current lowest distance to the master node. If so, the SN builds a new distance packet and transmits it to all its slave SNs, excepting those included in minimum-distance routes to the master node. This algorithm is similar to the *split horizon* algorithm [26].

[1] A *distance packet*: 186-bit DM1 packet ([17], pag. 60), carrying a 1-byte *distance field* (figure 2)

Therefore, the configuration process restarts in case of SN failures, and propagates changes from the failure neighborhood (possibly only affecting a BLN region).

If a SN receives a distance packet, it must update its routing table. The routing table stores pairs of neighbor SN addresses and their distances to the master node, and is sorted by distance. Thus, the best path to the master node is always on top of the table.

Let us show an example. Figure 3 depicts a BLN region. We describe two configuration steps to illustrate the general procedure.

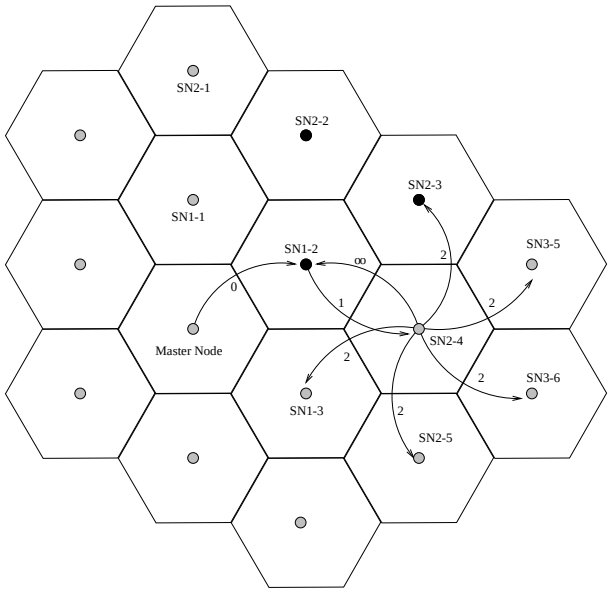


Fig. 3. BLN configuration

1. Once the master node initiates the process, SN1-2 distance to the master node changes. Node SN1-2 transmits the distance change to all its slaves.
2. SN2-4 receives the distance packet from SN1-2. Since this distance is lower than the current minimum distance, SN2-4 increments the received distance by 1, stores it in its routing table, and sorts the table again. For example, Table 1 shows both the original routing table of node SN2-4 and the new table contents.

As we said previously, if a SN detects that the minimum distance to the master node has changed, it builds a new distance packet with this minimum distance, and transmits it to its slaves (neighbors) except to those who are in the

Original	
Next Hop	Distance
SN1-2	∞
SN1-3	∞
SN2-5	∞
SN3-5	∞
SN3-6	∞

New	
Next Hop	Distance
SN1-2	2
SN1-3	∞
SN2-5	∞
SN3-5	∞
SN3-6	∞

Table 1. Original and new routing tables of SN2-4

minimum distance path (in our example, SN1-2). Those SNs receive a distance packet with infinite value, to prevent loops.

Remark 1: As far as a SN has less than seven neighbors less than 10 m away, it is possible to implement transmission links between all of them (the seven-slave transmission constraint of Bluetooth holds). This is valid for hexagonal-tiling, mesh and k-ary 3-cube BLNs.

Remark 2: A simple authentication handshake avoids connection establishment with invalid Bluetooth modems, which are considered invalid badges for simplicity. Typically, invalid badges will answer inquiry cycles with FHS packets, which is relatively harmless (see section 2.3 below). However, in case they answered with another kind of packets, they would be easily detected by the authentication handshake and rejected.

Remark 3: Badges do not try to establish data connections with SNs. They simply answer inquiries with FHS packets, which does not violate the seven-slave constraint.

2.3 BLN location protocol

The main goal of BLN is to track user movements in the target area. To meet that goal, all SNs have to send inquiries and collect badge responses. Every SN has a cache where it stores badge addresses. When it detects an inquiry response from a badge whose address was not in the cache, it builds a 366-bit DM1 *location packet* (figure 4), and transmits it to the SN on top of its routing table.

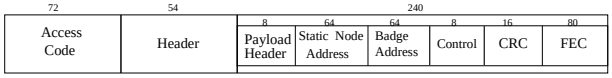


Fig. 4. Location packet format

For example, let table 2 be the current SN cache state (BD-X identifies the badge with address X) when the SN is performing an inquiry cycle. Before the cycle starts, the "*detected*" and "*new*" columns are unmarked (set to NO).

Bluetooth address	Detected	New
BD-3	NO	NO
BD-7	NO	NO
BD-11	NO	NO
BD-13	NO	NO
BD-17	NO	NO

Table 2. SN cache before the inquiry cycle

When a badge detects an inquiry, it answers with a FHS packet. The SN extracts the badge address from the FHS packet and checks it in the cache. If the address is already listed, the corresponding *detected* column is marked (set to YES). Otherwise, a new row with the address is added and both the *detected* and *new* columns are marked with YES. When the inquiry cycle ends, (i) all marked (YES) *new* columns are switched to unmarked (NO) state, and location packets for the corresponding entries are transmitted to the master node to report that new badges have entered SN range. (ii) All entries with unmarked (NO) *detected* column are deleted, and a location packet for each one of them is transmitted to the master node to report that the corresponding badges are now out of SN range.

Location packets carry two Bluetooth addresses: SN address and badge address. The packets have a bit to report if the badge arrives to or leaves the cell.

It should be understood that the SN that detects a badge is in charge of building location packets. All SNs placed along the transmission path to the master node simply relay them to the SN on top of their routing tables.

For example, table 3 represents a possible SN cache state after the inquiry cycle. Badge BD-19 was detected. Consequently, a location packet with BD-19 payload will be sent to the master node, and its *new* column will be changed to NO. Two badges, BD-7 and BD-11, have left the cell, and the corresponding location packets will be sent to the master node with the *detected* bit set to 0. Their entries will be removed. Badges BD-3, BD-13 and BD-17 are still around, but do not trigger location packet transmission.

Remark 4: SN responses to SN inquiry cycles are ignored by the location protocol, because the corresponding SNs are either listed in the routing table of the requesting SN or will be listed after establishing a master-slave link (this may happen, for example, when a dead SN is replaced).

Bluetooth address	Detected	New
BD-3	YES	NO
BD-7	NO	NO
BD-11	NO	NO
BD-13	YES	NO
BD-17	YES	NO
BD-19	YES	YES

Table 3. SN cache after the inquiry cycle

Remark 5: Obviously, invalid badges will answer to SN inquiries, and will generate location packets. However, those packets will be filtered by the master node. As we will see later, even if a large target area is crowded, BLN can carry all location packets, valid or invalid, as far as the number of users and the number of badges are roughly similar. Moreover, note that, if invalid badges correspond to static devices (such as printers), they will generate only *one* location packet, because their *new* column in SN location caches will be unmarked afterwards.

2.4 Location zones

The master node (or a service server attached to it) estimates that badge x is placed in a *location zone* that depends on the SNs that send location packets containing address x. Room-scale precision may support typical context-driven services in the scenarios in section 1, while keeping SN complexity reasonably low. We do not take signal strength nor signal delay into consideration, so far.

Location precision depends on the number of SNs that detect a given badge. We define different location zone *classes*, for a given network topology. A static badge is said to be located in a class I zone of a hexagonal tiling topology if *at most* three SNs detect the badge. For example, the class I zone depicted in figure 5 is defined by SN2-2, SN2-3 and SN3-3 detection. Note that neither SN1-2, SN3-2 nor SN3-4 detect the badges in that zone. A class I zone has an area of 16.12 m^2. A badge is said to be located in a class II zone if at most four SNs detect it. For example, SN2-2, SN2-3, SN3-2 and SN3-3 define the class II zone in figure 5. Class II zones have an area of 18.12 m^2.

Class III zones are special cases in the neighborhood of SN failures (or at the edges of the target area). The class III zone in figure 5 has an area of 34.24 m^2. In absence of SN failures, those zones can be avoided by placing extra SNs at the walls of the target area.

Finally, a class IV zone is a single SN. When a badge is placed exactly at a SN position, it is detected by seven SNs (the closest SN and all its slaves).

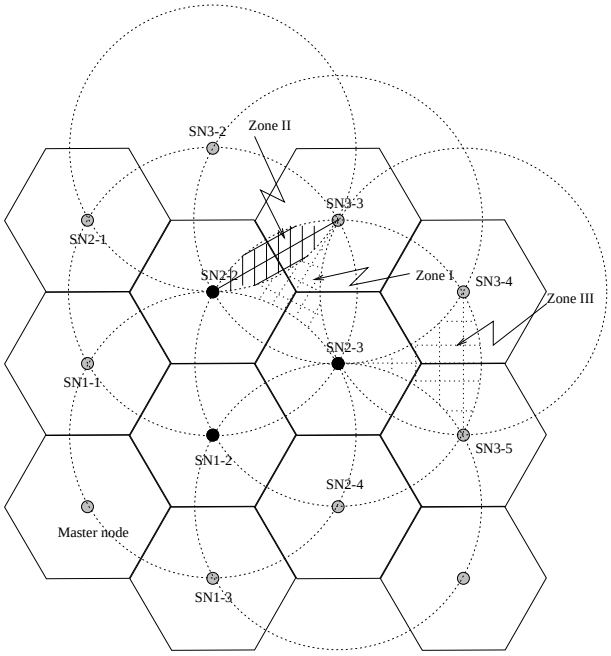


Fig. 5. Location zones

2.5 Scalability

As BLN grows, the number of location packet hops may become too large, if originated at BLN edges. It is possible to avoid this issue by installing additional master nodes. Due to BLN symmetry, when the configuration protocol converges to a equilibrium point, the maximum distance to each of the master nodes is similar, if equally spaced. Figure 6 shows an example for four master nodes.

Note that the maximum distance route to any master node has three hops. However, even for the same network, location routes could change at the next initialization, since all SNs evolve asynchronously. In order to evaluate BLN symmetry, we performed simulations of several three-layer ($\sim$ 4,000 m^2) BLN initializations. The results are summarized in table 4. In it, MN is the number of master nodes, $\overline{SN}_{MN}$ is the average number of SNs per master node (their minimum distance routes lead to the same master node) and $\overline{d}_{MN}$ is the average maximum distance per master node.

Note that, spontaneously, the area managed by each SN is inversely proportional to their number, and the maximum distance per master node is quite stable across different initializations.

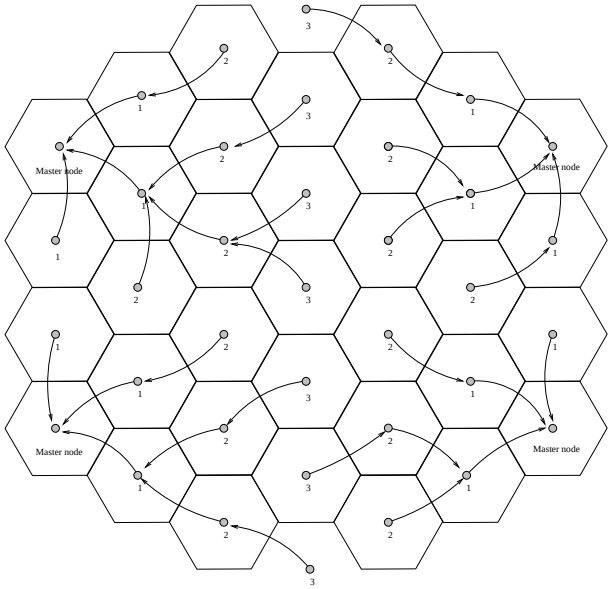


Fig. 6. Location packet routes (sample), four master nodes

MN	$\overline{SN}_{MN}$	$\overline{d}_{MN}$
1	36	2.3333
2	17.5	2.54 + (-0.06,0.09)
4	8.25	1.7134 + (-0.21,0.38)

Table 4. BLN symmetry evaluation

3 Survivability

3.1 BLN reconfiguration

BLN detection capabilities survive to failures, due to spontaneous reconfiguration when a SN dies, which keeps as many SNs connected to the master node as possible. Survivability is inherent to BLN configuration protocol. Suppose SN a does not respond to an inquiry from SN b. If, as a consequence, the minimum distance changes, b transmits its new minimum distance to its slaves. The change in minimum distance propagates to outer layers.

As an example, table 5 shows the evolution of the routing table of SN2-1, when all SNs in the first layer go down successively.

Only after all SNs in the first layer die, SN2-1 will be isolated from the master node.

SN2-1 routing table													
Beginning		SN1-2 down		SN1-1 down		SN1-6 down		SN1-5 down		SN1-4 down		SN1-3 down	
Route	Dist.	Route	Dist.	Route	Dist.	Route	Dist.	Route	Dist.	Route	Dist.	Route	Dist.
SN1-1	2	SN1-1	2	SN2-12	3	SN2-2	5	SN2-2	5	SN2-2	5	SN2-2	∞
SN2-2	3	SN2-2	3	SN3-18	4	SN3-2	6	SN3-2	6	SN3-2	6	SN3-2	∞
SN2-12	3	SN2-12	3	SN2-2	∞	SN3-18	6	SN3-18	∞	SN3-18	∞	SN3-18	∞
SN3-18	∞	SN3-18	∞	SN1-1	∞	SN3-1	∞	SN3-1	∞	SN3-1	∞	SN3-1	∞
SN3-1	∞	SN3-1	∞	SN3-1	∞	SN2-12	∞	SN2-12	∞	SN2-12	∞	SN2-12	∞
SN3-2	∞	SN3-2	∞	SN3-2	∞	SN1-1	∞	SN1-1	∞	SN1-1	∞	SN1-1	∞

Table 5. Routing table of SN2-1

3.2 Badge detection survivability

In section 2.4 we saw that badge position cannot be known with more precision than the area of the location zone where it belongs. However, as SNs go down, it could happen that a badge crosses a region of the target area undetected. We made simulations on a three-layer BLN to evaluate survivability of BLN detection capabilities in multiple failure scenarios. Figure 7 shows the results, for a single, two or four master nodes (when there are more than one master node, they are equally spaced at target area edges). The simulations were made for a quality of 90 % and a tolerance of 5 %, using the Batch Means Method [27].

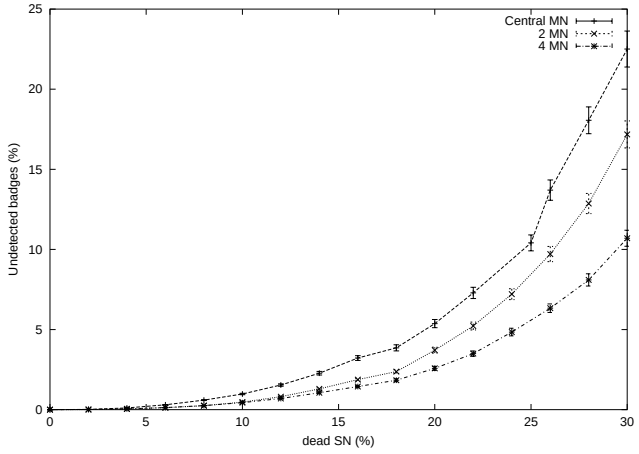


Fig. 7. Undetected badges

In our simulation, master nodes are protected against failure. This is certainly realistic, since the whole BLN relies on them. It is important to mention that, in a real scenario, no more than 5%-10% of the SNs should be ignored if dead (or, in other words, if they are maintained as frequently as light bulbs. A percentage of 30% dead SNs means that target area is probably close to bankruptcy!). Note that, for as many as 10% dead SNs, only 1% of the badges were undetected.

3.3 Isolated SNs in case of failure

As another survivability measure, we evaluated the percentage of SNs that must die to isolate a single operational SN (it is impossible to establish a path between that SN and the master node), for a single master node. Figure 8 shows the simulation results. Note that, in the three-layer BLN of the previous example, more than 20% of the SNs should die to isolate a single operational SN.

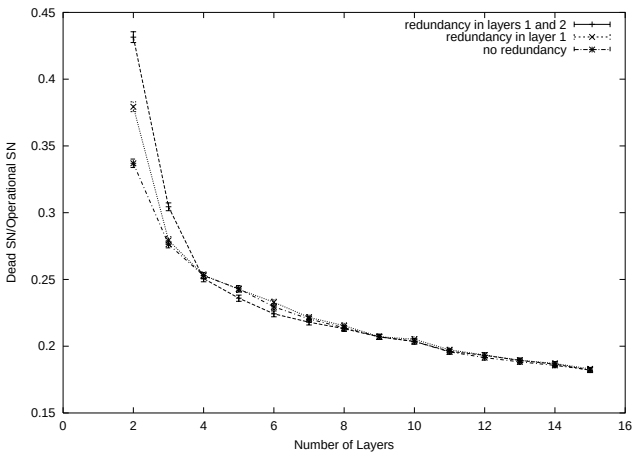


Fig. 8. Dead SNs to isolate a single operational SN

4 Conclusions

In this paper, we have proposed an auxiliary Bluetooth location network for context-driven services, BLN. The location network has the following characteristics:

- BLN carries position information to the servers without user participation.
- BLN is based on a general-purpose RF technology, available as an option in existing handhelds.
- BLN can be used as an emergency data network.

- BLN infrastructure consists of small, completely independent Bluetooth nodes (no wires).
- Spontaneous topology configuration allows scalability, by placing as many master nodes as necessary.
- BLN can coexist with Bluetooth devices that are not part of the location system, such as printers or headphones.
- BLN has self-diagnosis capability. SN failure does not compromise BLN operation (i.e. maintenance is not critical). The hexagonal-tiling BLN admits failure of approximately 10% SNs, which implies less precision, but all badges are still detected. In case of multiple SN failures, BLN reconfigures itself spontaneously.

Forthcoming work will evaluate BLN performance and study alternative BLN topologies: mesh and k-ary 3-cube BLNs.

References

1. m-commerce world. WWW. http://www.m-commerceworld.com/.
2. U. Varshney, R. J. Vetter, and R. Kalakota. Mobile commerce: A new frontier. *Computer*, 10:32–38, 2000.
3. A. Darling. Waiting for the m-commerce explosion. *Telecommunication International*, 3:34–39, 2001.
4. Datamonitor. m-commerce infraestructure in the u.s. WWW. http://www.datamonitor.com.
5. Electronic Guidebook Project. WWW. http://www.exploratorium.edu/guidebook.
6. Bluetooth iPAQ. WWW. http://www.compaq.com/products/handhelds/pocketpc/options/expansion_packs.html.
7. Bluetooth Nokia. WWW. http://www.nokia.com/phones/6210/bluetooth.html.
8. ETSI document TS 122 071 V3.2.0.
9. A. Dorman. Can m-commerce find a place in your network? WWW. Network Magazine. http://www.networkmagazine.com/article/NMG20011101S0005/2.
10. J. Werb and C. Lanzl. A positioning system for finding things indoors. *IEEE Spectrum*, 35(9):71–78, 1998.
11. T. Kindberg and J. Barton. A web-based nomadic computing system. *Computer Networks*, 35(4):443–456, 2001.
12. G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: a mobile context-aware tour guide. *Wireless Netwoks*, 3(5):421–433, 1997.
13. R. Want, W. N. Schilit, N. I. Adams, R. Gold, K. Peterson, D. Goldberg, J. R. Ellis, and M. Weiser. *The ParcTab Ubiquitous Computing Experience*. In *Mobile Computing*. Kluwer Academic Publishers, 1996.
14. N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket location-support system. In *Proc. of the Sixth Annual ACM Internation Conference on Mobile Computing and Networking*, 2000.
15. A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. In *Proc. of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, 59–68, 1999.

16. J. García-Reinoso, J. Vales-Alonso, F. J. González-Castaño, L. Anido-Rifón and P. S. Rodríguez-Hernández. A new m-commerce concept: m-mall. *Lecture Notes in Computer Science*, 2232:14–25, 2001.
17. Bluetooth SIG. Specification of the Bluetooth system-core v1.0b. Technical report, December 1999.
18. 10meters.com. WWW. http://www.10meters.com/blue_802.html.
19. J. Lansford and P. Bahl. The design and implementation of HomeRF: A radio frequency wireless networking standard for the connected home. *Proceedings of the IEEE*, 88:1662–1676, 2000.
20. GSMBox. WWW. http://uk.gsmbox.com/news/mobile_news/all/19304.gsmbox.
21. Journaldunet. WWW. http://solutions.journaldunet.com/0103/010307inventel.shtml.
22. Security-informer. WWW. http://www.security-informer.com/english/crd_bluecore01_209282.html.
23. Casira. WWW. http://www.csr.com.
24. SPIKE. WWW. http://www.spike-wireless.com/main.html.
25. W. Mao and D. M. Nicol. On k-ary n-cubes: theory and applications. NASA CR-194996 ICASE report # 94-88, 1994.
26. A. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
27. A. M. Law and J. S. Carson. A sequential procedure for determining the length of a steady-state simulation. *Operations Research*, 27:1011–1025, 1979.
28. Bluetooth discussion list. WWW. http://www-124.ibm.com/pipermail/bluehoc-discussion/.

Dynamic Query Execution in Distributed Database Systems

Josep M. Muixi[1], August Climent[2], Miquel Bertran[2], Miquel Nicolau[2], and Francesc Babot[2]

[1] IT Architecture Department, GFT Iberia Consulting, Sant Cugat del Valles, Barcelona, SPAIN
josep-m.muixi@db.com,
[2] Computer Science Department, Enginyeria i Arquitectura La Salle, Ramon Llull University, Barcelona, SPAIN
{augc,miqbe,miqueln,fbabot}@salleURL.edu

Abstract. Distribution offers advantages and new possibilities to traditionally centralized environments; databases are no exception, and Distributed Database Management Systems (DDBMS) are the direct outcome. But distribution does not imply immediate improvements necessarily: some new concepts and issues need to be considered then, as availability, distributed security or load balancing. Focus in this article will be on load balancing DDBMS environments; some techniques have been presented in the past and a new mechanism is to be introduced in this article. Particularities of this new load balancing mechanism, as no centralized component or dependency, and its probabilistic nature form the basis of new load balancing technologies. Public standards for OLTP environments performance measurement have been followed in reporting results presented.

1 Introduction

DDBMS architectures can become complex to be defined, administered and managed. They have been studied and categorized in previous literature [1]. These architectures allow more independent and local control of data and provide easy alternatives in designing and upgrading the distributed system.

Data availability is increased by *Replication* methods [2] applied to distributed architectures. Globally improved performance is obtained at the expense of some extra handling logic; *replication* is very common in currently available systems, representing a significant benefit on most of cases.

Query planning —including *optimization* and *scheduling*— for Database Management System (**DBMS**) operations [3] and their application to **DDBMS**s has been discussed extensively in literature [4]. *Optimization* and *scheduling* act as key pieces of load balancing strategies, as from their direct application, load is finally distributed in the system.

Parallelism adoption in Database Management Systems (**DBMS**s) helps to increase performance in a high degree under normal circumstances and its use is a must.

Once at this point, further research was dedicated to studying the application of *pipelining* techniques to these *parallel* systems [5].

The general concept of *workload balance* for distributed systems has been studied on many occasions in the past, focused mainly on the idea of obtaining a scheduling plan of tasks over a set of CPUs given a generic operation to be executed over this architecture [6], but *workload balance* for **DBMS**s is an issue which has been addressed in fewer cases in the past [7], when attention has been centered mostly on query processing optimization.

[1] presents an introduction to *workload balancing problem* in the Distributed Database Management System context. Several solutions have been proposed for multiprocessor machines [8] and distributed systems [7].

The problem addressed is the distribution of relational operations among nodes of a **DDBMS**, in a way that the load is balanced from a global view of the system; this is done in order to get a better performance of the **DDBMS** and to get a normalized and efficient use of the **DDBMS** resources as described in [5].

Load balancing strategies historically have been focused on centralized methodologies. Considering this, we prepared a set of simulations based on *DDBSim*Tool[1]: a representative set of centralized *workload balancing* policies were abstracted and simulated, and finally compared with the simulation results obtained using the distributed Just-in-Time strategy introduced.

Most current *query planning* mechanisms have been designed to obtain optimal query executions with no *workload balancing* motivation on them. Moreover, when *workload balancing* is addressed, it is done from a point of view absolutely apart from *query planning*. *Load balancing* techniques usually use a centralized component or information repository. The method presented here breaks this pattern and introduces a real distributed *load balancing* method integrated with *query planning* and *query execution* phases of distributed query order executions.

2 Basic Workload Balance strategies

Basic workload balance strategies applied to **DDBMS**s can be classified in these five categories [5]:

No load-balancing (No). This is current standard adopted in many commercial systems. Normally the queries are processed under a predefined plan and assigned for execution in a static form.

Round Robin (RR). Load is assigned to the next node listed in a circular list. Easy to implement, it is probably the most used strategy because of its high revenues compared to the minimum effort required in its implementation.

Join-the-Shortest-Queue (SQ). Incoming load is assigned to those nodes with shortest queues of tasks to process.

[1] *DDBSim* is a simulator for Distributed Database Management System architectures, developed with the main objective to get experimental performance results.

Join-the-Fastest: Processor Utilization (Pr). Processors showing the lowest utilization will be the next candidates to get incoming load.

Join-the-Fastest: Response Time (Rt). Incoming load is assigned to nodes which recently showed the fastest response times servicing previous requests.

3 Just-in-Time Workload Balancing

In this section our new concept of workload balancing policy is introduced.

3.1 Context and motivations

From previous work [7], we can conclude that a balanced system is one which delivers high performance; the less balanced a system is, the less performance we could expect from it. The main objective of this study will be, therefore, trying to find new methods to optimize, or at least improve, system load balance for **DDBMS** environments.

The method shown here employs a new philosophy in scheduling operations in a **DDBMS**. This method is based on previous studies for multiprocessor environments where database query executions are modelled as hierarchical trees of operations [9]; applying the same model for distributed query executions, and using new mechanisms defined in this article, an effective and improved load balancing policy can be achieved.

The proposed strategy for database operations scheduling is based on the fact that operation executions can be carried out in more than one node[2]of the distributed architecture; normally this is achieved with the help of data *replication*.

In general, scheduling for queries to be executed over any database is done in a near-static or fixed form, or based on some dynamic information about current status of the database, data location, resource availability, etc. Problems with the first approach appear as no load distribution exists over replicated data and distributed resources. The second approach is closely related to final mechanisms and techniques used in the application of the method; anyway, using dynamicity has been tied historically to the figure of a global scheduler or manager module which centralizes some system status information; this approach is clearly better than the static one, but also has some disadvantages: the system overload generated by the need to have a centralized component updated with system resources statistics and status information, and the unpredictable evolution in time of system load, can turn the best scheduling approach at scheduling time t_{schd} in to the worst plan execution at execution time t_{exec}.

As we aim to show load balancing techniques and their contribution to system performance, we will rely our study on load metrics for distributed environments

[2] Depending on the distributed architecture, nodes will be composed or understood differently; in this work, a node is assumed as the set of components able to execute an operation of a distributed query plan, sharing or not resources.

presented in 3.2. Being the final intention of any load balancing strategy to improve system performance, we present a performance metric in 3.2.

3.2 Previous concepts

Some concepts used afterwards should be introduced before exposing new workload balance strategy.

Bushy trees. Database query executions can be modelled as graphical representations of their composing operations. Each graphic represents a query decomposition into operations: each node inside this graphic defines an operation composing the main query; nodes with no children may be scans, sorts or merges; nodes with two children generally represent joins, but can be extended to any binary relational operation [1] supported by the **DDBMS**. These graphical representations of query operations are called *bushy trees*.

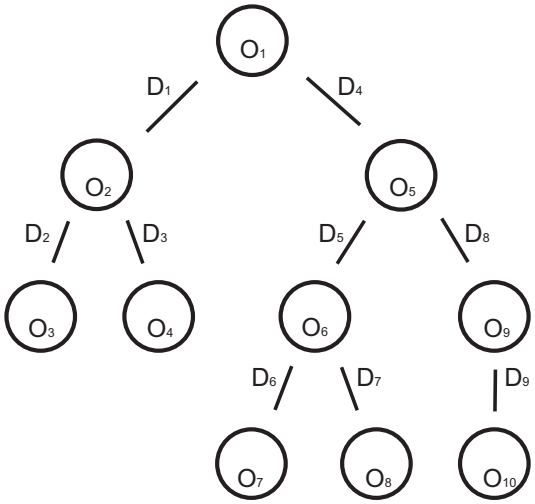


Fig. 1. *Bushy-tree* sample.

In general, a *bushy tree* $\mathbb{B}$ is composed of a set of nodes O_i representing operations, and a set of edges D_j representing dependencies as represented in *figure 1*, this is:

$$\mathbb{B} = (\mathbf{O}, \mathbf{D})$$

Scheduling operations . Once the logical decomposition of a query has been defined in the form of a *bushy tree*, a distribution or scheduling of the resulting

operations to processors or nodes has to be specified, a problem known as the *processor partitioning problem* in parallel contexts. Studies and optimizations have been presented to get the right composition and distribution for operations in multi-processor environments [9] and special methodologies exist for distributed architectures [6].

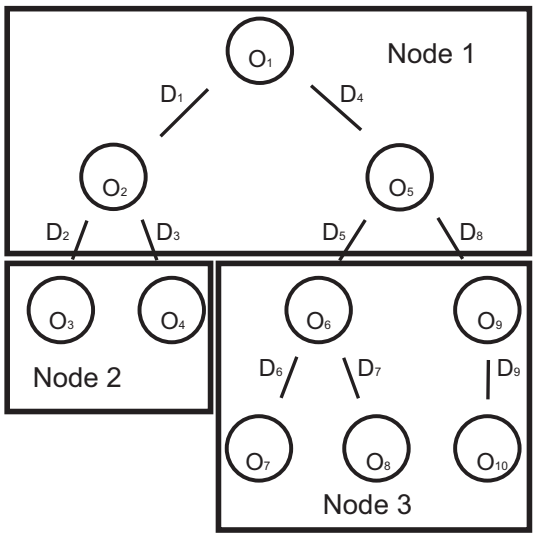


Fig. 2. *Bushy-tree* operations allocation.

Load Metrics. In [10] some load metrics and their interpretation are presented. *Imbalance* and *Dynamicity* are the two metrics used in our comparison section.

Imbalance Standard deviation (σ) of the load of the N system nodes, normalized to the average system load (λ).

$$\sigma = \frac{1}{\lambda}\sqrt{\frac{\sum_{i=1}^{N}(L_i - \lambda)^2}{N}} \tag{1}$$

with

$$\lambda = \frac{1}{N}\sum_{i=1}^{N}(L_i)$$

being L_i the load of the ith system node

Dynamicity Normalized standard deviation of the load changes between time t and $t + \Delta t$.

$$\Delta\sigma(t, t + \Delta t) = \frac{1}{\lambda(t)} \sqrt{\sum_{i=1}^{N} \frac{\left(\Delta L_i(t, t + \Delta t)\right)^2}{N}} \quad (2)$$

being $\lambda(t)$ the average system load at instant t.

Please refer to [10] for further conclusions and details about these metrics.

TPC Benchmark^{TM} C. TPC Benchmark^{TM} C (**TPC-C**) is a performance metric for On-Line Transaction Processing (**OLTP**) systems defined by the *Transaction Performance Council*[3].

Performance metric reported by **TPC-C** is a throughput that measures the number of orders processed per minute or *tpmC* once a standard load has been applied to a **DBMS**. This is the preferred performance metric used in results section.

3.3 JitWob method

The *Workload balancing* strategies introduced in section 2 on page 3 form a basic range of mechanisms to provide load balancing; some other techniques have appeared with alternative strategies making use of ideas from other fields of knowledge unrelated to computer science such as economics [11].

SQ, **Pr** and **Rt** methods are based on the assumption of load information availability: current load information of the whole system should be accessible in order to take a decision to balance the overall system load. Only two possibilities are available to make this approach work: schedule and launch all queries from a centralized component which then has direct access to load information; or send the load information from all nodes to the decentralized component which is going to do the scheduling. Any variant of these two solutions is also feasible.

Anyway, both of these possible solutions to system load information availability tend to present problems, involving: bottlenecks, network and nodes overload, updating delays...

The strategies mentioned up to now are reacting on current load information of the system, but not Just-in-Time: scheduling policy is based on collected system load information updated periodically. Effectiveness of commented methods depend on relation: *system information collection rate versus system load variation rate.*

Just-in-Time Workload Balancing (*JitWob*) evolves the query execution process and goes one step ahead, splitting *load balancing* logic into *scheduling* and *execution* phases. The concept of *execution* phase being considered typically

[3] TPC Benchmark^{TM}, TPC-C^{TM}, and tmpC^{TM} are trademarks of the Transaction Performance Council (*http://www.tpc.org*).

a passive phase disappears with *JitWob*; now workload balancing logic is also applied at execution time: it is a *reactive load balancing* policy.

JitWob does not collect any information about real-time load of the system, as no centralized point exists for such a task; on the other hand, the system is able to react to balance its load once operations have been distributed among required nodes —at runtime—, providing Just-in-Time *load balancing*; this main idea is what is new from the point of view of *optimization*, *scheduling* and *execution* for database queries: a distributed architecture using a real distributed algorithm to create and adapt execution plans dynamically just-in-time according to system load.

Under *JitWob*, each query-generating node acts independently as an *optimizing* and *scheduling* engine, allocating operations for the final execution of a query operation to all required nodes of the distributed architecture, in the same way it could be implemented following any of the existing scheduling strategies. If some statistical information is needed it can be collected from a statistics repository if the *scheduling* algorithm being used requires it, as is being done in existing *scheduling* algorithms. *JitWob* acts by balancing system load at runtime, when operations have been sent to nodes and 'just-in-time' load is found; at this point the *JitWob* algorithm can choose between accepting the execution of that operation in that node, wait for a lower load on that node, or re-schedule it to another node.

JitWob works by complementing existing *scheduling* algorithms, acting at runtime. The only requirement *JitWob* establishes to the scheduling algorithm is to specify alternative nodes where execution for that operation can be done[4]; this will make the task of *JitWob* easier at runtime if the operation has to be re-allocated to an alternative node.

Method theory. Under *JitWob*, the *optimizing* phase of any query is managed in the same way as usual: an optimization algorithm is responsible for creating the best operations map to execute the query; this map can be represented as a *bushy tree* $\mathbb{B}$ of operations.

The *scheduling* phase of the query execution is to create the allocation plan for each one of the operations O_i composing graph $\mathbb{B}$ among the different nodes composing the **DDBMS** architecture considering D_j dependencies. All of these algorithms manage *execution dependency*, *system fragmentation* and *replication* as main parameters to define final nodes allocation, under the normal assumption of *synchronous execution time*; for further details, please refer to [6]. The final result of this process is an allocation table $\mathbb{A} = (A_i)$ with as many elements as nodes appear in the *bushy tree* $\mathbb{B}$ created in the previous phase; such a table assigns an operation O_i to a node in the distributed database $N_{opt(i)}$ where the execution should be optimal, so that:

$$\mathbb{A} = (A_1, A_2, \ldots A_n) \quad | \quad \forall\, A_i \quad \exists \quad (O_i \rightarrow N_{opt(i)})$$

n being the number of operations in the *bushy tree* obtained.

[4] this normally will imply data replication across the distributed architecture.

JitWob proposes only a subtle modification to these algorithms creating such a scheduling plan; at scheduling time, the final allocation table $\mathbb{A}'$ should include not only the best target node for its execution $N_{opt(i)}$, but also alternative node locations ordered by preference $N_{alt(1)}, N_{alt(2)}, ...N_{alt(k)}$, where execution of that task is also feasible. The final relation created during scheduling phase could then be:

$$\mathbb{A}' = (A'_1, A'_2, \dots A'_n) \quad | \quad \forall\ A'_i \quad \exists \quad R_i$$

where R_i is the relation:

$$O_i \rightarrow N_{opt(i)},\ N_{alt(1)},\ N_{alt(2)},\ \dots\ N_{alt(k)}$$

k being the *JitWob* depth with a maximum value of N–1, with N as the number of nodes in the **DDBMS**. A sample of this new kind of allocation relation is shown in table 1.

The intention of this modification is that if an overload is detected in any of the nodes $N_{opt(i)}$ participating in the query execution at runtime —once each operation has been allocated or distributed to each node— *JitWob* will be able to choose a different node from relation A'_i, trying to find the best alternative in each case. This being the main idea behind new proposed *JitWob* methodology, some variants can be found depending on the way of selecting alternative nodes. Here is where *Probability Theory* starts its role, and let us apply different *JitWob* policies according to the different probabilistic distributions used in selecting an alternative node for allocation —this is applied at scheduling time as well as runtime if reallocation is needed due to local overload—. Results presented here are obtained under a *gaussian probabilistic distribution*; so this is the preferred strategy used in our simulations. *Gaussian probabilistic distribution* is applied to current set of alternate nodes where current operation can be alternatively executed, assigning more probability to nodes considered more appropriate by the scheduling mechanism; once probability laws have elected the new node based on this distribution, execution of current operation is transferred there.

Table 1 is an example of an allocation table after the *scheduling* phase for a *JitWob* compliant query processing system; alternative nodes are specified in order of preference as possible locations for operation execution.

4 JitWob results

4.1 Simulation parameters

Architecture basic characteristics of **DDBMS** used in simulations are summarized in table 2. This is the reference architecture this study is based on; it is composed of 128 server nodes composing the **DDBMS** core. Generic clients with characteristics summarized in table 2 where simulated. The amounts of clients used determined high loads in the server side and enabled tested load balancing techniques to show their effects.

Operation	**Optimal Candidate Node** N_{opt}	**1st Alt. Candidate Node** $N_{alt(1)}$	**2nd Alt. Candidate Node** $N_{alt(2)}$
O_1	N_1	N_2	N_3
O_2	N_1	N_2	N_3
O_3	N_2	N_3	N_1
O_4	N_2	N_1	–
O_5	N_1	–	–
O_6	N_3	N_2	–
O_7	N_3	N_1	N_2
O_8	N_3	N_1	N_2
O_9	N_3	N_2	–
O_{10}	N_3	N_2	N_1

Table 1. *Sample of* JitWob *operations allocation table A' for bushy tree of figure 1 as defined by the optimal allocation graph depicted in figure 2 with level of depth $k = 2$ and number of Nodes $N = 3$.*

System Components	**Clients (× 280,000)**		**Servers (× 128)**	
	Qtty	Description	Qtty	Description
Processor	2	Pentium III @ 733MHz	8	Pentium III @ 900MHz
Cache	1	256KB	1	2MB
Memory	4	128MB	32	512MB
Disk Drives	1	9.1GB	25	12GB
Network	100Mbits		1Gbits	

Table 2. *Main parameters describing simulated architecture.*

4.2 JitWob compared

Simulation results are presented in two forms:

Graphical load results: Show system *imbalance* and *dynamicity* for a period of 10 minutes in a stable period of load, exactly in the time-frame compressed between 5 minutes after measurement interval starts and 5 minutes before measurement interval ends.

Performance summary: Table 3 shows performance results in *tpmC* units as defined in **TPC-C** standard for a variety of client configurations.

No Load Balance policy is the easiest technique where destination nodes for operations of a query are predefined and have no variation over time: it is the simplest policy to implement and one of the most common found in the real world. The main reason for distributing operations (if distributed) is data availability, load balance having no significant weight on final decision of scheduling

algorithm. Performance results shown in table 3 offer the lowest performance results in any of the simulated cases.

The *Round Robin* policy works very well in real cases and is the most implemented load balancing strategy on the market. Its effectiveness depends on the homogeneity of operations and system nodes; another advantage of this policy is its independence of global system load information. Under *Round Robin* policy, *imbalance* is kept low, as well as *dynamicity*, but this one suffers some high peaks; these peaks are mainly caused because load distribution is not based on system status and some localized overloads can appear. *Dynamicity* and *imbalance* indexes keep similar values. *System Load* under extreme query load is about 80%, notably higher than the previous results (30%) obtained under *No Load Balancing* policy; accordingly, the performance results obtained using this policy improve substantially the ones got with previous one.

Number	**Load Balancing strategy**			
of users	No	RR	SQ	JitWob
35,000	12,121.5	14,527.2	15,744.8	15,792.7
32,000	16,143.8	19,155.2	20,143.0	20,342.1
30,000	18,497.1	25,934.9	29,720.8	30,376.2
28,800	**19,312.9**	**28,353.0**	**34,221.0**	**36,312.9**
25,000	18,262.3	26,002.4	29,212.8	31,361.0
22,000	17,972.6	24,324.0	27,557.0	28,612.4
20,000	16,173.5	21,921.2	23,129.0	23,341.3
15,000	10,372.7	12,254.0	12,173.9	12,167.1

Table 3. *Simulation performance results in* tpmC.

Applying *Shortest Queue* policy an improvement over previous strategy is obtained. *Imbalance* is reduced in this case and *dynamicity* is sightly superior to it. System Load gets values of almost 90%; performance results are again outperforming previous strategy ones.

Simulation results for *JitWob* policy appear in figure 3. *Imbalance* and *dynamicity* get lower values than in previous cases —indicating a more balanced and stable system—; what is remarkable is the fact that no peak in its values is superior to peaks obtained in previous methods in percentage. *System Load* measure shows a system taking advantage of all its resources in a more stable form through time, which makes final performance results outperform other strategies (see table 3).

5 Conclusions

JitWob has been the most effective *load balancing* strategy and the one which, overall, has achieved the greatest improvement in performance. It is clear that

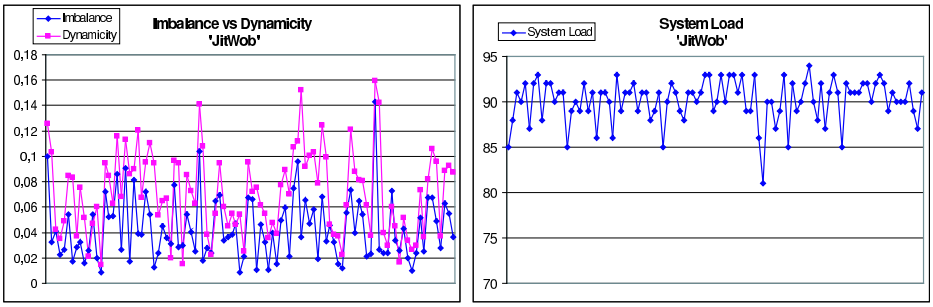


Fig. 3. Simulation load results for *JitWob* Strategy.

overhead caused by *JitWob* policy is minimum, so no performance overload can be considered from its use. In case of no replicated or no overloaded databases, *JitWob* adds no exceptional value to execution of query operations. On the other hand, if replication and overloading of some nodes is probable, then *JitWob* acts as the most suitable and easy-to-implement solution. Its decentralized and probabilistic philosophy provide robustness and independence to distributed nodes, outperforming in load balance performance.

JitWob simplicity, compared to other *workload balancing* strategies, make it a true candidate for real systems.

6 Further steps

What stands *JitWob* as a good alternative to other *workload balancing* strategies —apart from its better performance— is its simpler implementation, less runtime load, and robustness —no need for central components or statistics—. Future work is to go in the direction of study of this inherent characteristics of the innovative load balancing strategy; its distributed and probabilistic nature form a basis for a deeper analysis of its effects and characteristics exploitation. Its adoption in non-relational environments also opens new research lines.

References

[1] M. Tamer Ozsu and Patrik Valduriez, *Principles of Distributed Database Systems* (New Jersey: Prentice Hall International, 1999).

[2] M. Nicola and M. Jarke, Performance Modeling of Distributed and Replicated Databases, *IEEE Transactions on Knowledge and Data Engineering*, 12(4), 2000, 645-672.

[3] Ming-Syan Chen and Philip S. Yu, Optimization of Parallel Execution for Multi-Join Queries, *IEEE Transactions on Knowledge and Data Engineering*, 8(3), 1996, 416-428.

[4] Salvatore T. March and SangKyu Rho, Allocating Data and Operations to Nodes in Distributed Database Design, *IEEE Transactions on Knowledge and Data Engineering*, 7(2), 1995, 305-317.

[5] Mahdi Abdelguerfi and Kam-Fai Wong, *Parallel Database Techniques* (Los Alamitos, California: IEEE Computer Society Press, 1998).

[6] Jan Jonsson and Jonas Vassell, Evaluation and Comparison of Task Allocation and Scheduling Methods for Distributed Real-Time Systems, *Proc. 2nd IEEE Int. Conf. on Engineering of Complex Systems*, Lyon, France, 1996, 226-229.

[7] Erhard Rahm, Dynamic Load Balancing in Parallel Database Systems, *Proc. EURO-PAR 96 Conf. LNCS*, Lyon, France, 1996, 37-52.

[8] Ming-Ling Lo and Ming-Syan Chen, On Optimal Processor Allocation to Support Pipelined Hash Joins, *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of data*, Washington D.C., USA, 1993, 69-78.

[9] Chiang Lee and Chi-Sheng Shih, Optimizing Large Join Queries using a Graph-Based approach, *IEEE Transactions on Knowledge and Data Engineering*, 13(2), 2001, 298-315.

[10] Antonio Corradi and Letizia Leonardi, Diffusive Load-Balancing Policies for Dynamic Applications, *IEEE Concurrency*, 7(1), 1999, 22-31.

[11] M. Stonebraker and R. Devine, An economic paradigm for query processing and data migration in Mariposa, *Proc. 3rd. Int'l Symp. on Parallel and Distributed Information Systems*, Austin, TX, 1994, 58-67.

Algoritmo parametrizable que implementa memoria compartida distribuida con coherencias secuencial, causal y caché

Ernesto Jiménez[1], Antonio Fernández[2], and Vicente Cholvi[3]

[1] Universidad Politécnica de Madrid, 28031 Madrid
ernes@eui.upm.es
[2] Universidad Rey Juan Carlos, 28933 Móstoles, Madrid
afernandez@acm.org
[3] Universitat Jaume I, 12071 Castellón
vcholvi@inf.uji.es

Resumen En [6] hemos presentado un algoritmo que puede ser usado para implementar coherencia secuencial, causal o caché en sistemas de memoria compartida distribuida. En este artículo presentamos en forma resumida los principales resultados de dicho trabajo. Para completar los detalles, se puede acceder al trabajo original o contactar a los autores.

1 Introducción

La memoria compartida distribuida (MCD) es un mecanismo bien conocido para la interconexión de procesos en un entorno distribuido. Una de las propiedades más importantes de un sistema MCD es la semántica de sus operaciones de lectura y escritura, lo que se conoce comúnmente como su *modelo de coherencia*. Mientras que la semántica de las operaciones de lectura y escritura en un programa secuencial está clara, no ocurre lo mismo cuando se permiten accesos concurrentes a variables compartidas. Este problema se agudiza si la memoria compartida no está centralizada, sino que está distribuida entre varios procesadores (o sea, tenemos memoria compartida distribuida).

Dos de los modelos de coherencia propuestos más populares son el modelo secuencial [7] y el modelo causal. La semántica del primero es más cercana que la del segundo a lo que los programadores esperan de la memoria compartida, mientras que la semántica del segundo es más débil, pero permite implementaciones eficientes del mismo. Como consecuencia, en la literatura se han propuesto varios algoritmos que implementan coherencia secuencial [1,3,4] y coherencia causal [2,8,9]. Un tercer modelo de coherencia propuesto en la literatura es el caché [5], del cual (por lo que sabemos) no existía antes de nuestro trabajo ningún algoritmo propuesto.

Una propiedad interesante de cualquier algoritmo que implemente un modelo de coherencia dado es el tiempo que tarda en ejecutarse una operación de memoria (una lectura o una escritura). Si las operaciones no necesitan esperar a que finalice alguna comunicación por la red, y pueden completarse usando solamente

```
Initialization ::
begin
  turn_p ← 0
  updates_p ← ∅
end

r_p(x) :: atomic function
begin
  if (model = sequential) and
     (updates_p ≠ ∅) and
     ((x, ·) ∉ updates_p) then
       wait until turn_p = p
  return(x_p)
end

send_updates() :: atomic task activated
whenever turn_p = p
begin
  /* send to all processes, except itself */
  broadcast(updates_p)
  updates_p ← ∅
  turn_p ← (turn_p + 1) mod n
end

w_p(x)v :: atomic function
begin
  x_p ← v
  if ((x, ·) ∈ updates_p) then
       remove (x, ·) from updates_p
  include (x, v) in updates_p
end

apply_updates() :: atomic task activated
whenever turn_p = q, p ≠ q, and the set
updates_q from process q is in the receiving
buffer of process p
begin
  take updates_q from the receiving buffer
  while updates_q ≠ ∅ do
       extract (x, v) from updates_q
       if (model = causal) or
          ((x, ·) ∉ updates_p) then
            x_p ← v
  turn_p ← (turn_p + 1) mod n
end
```

Figura1. Algoritmo propuesto en [6]. Se presenta el código ejecutado por el proceso p. El algoritmo se invoca con un parámetro *model*, el cual define el modelo de coherencia que se debe implementar.

el estado local del proceso que las generó, se dice que son *rápidas*. Operaciones de memoria rápidas es una propiedad muy deseable en cualquier implementación de un modelo de coherencia. Todos los algoritmos de coherencia causal referenciados anteriormente son rápidos. Sin embargo, Attiya y Welch [3] han demostrado que no hay ninguna implementación de coherencia secuencial que garantice la ejecución rápida de todas sus operaciones. Este resultado de imposibilidad limita la eficiencia de cualquier implementación de coherencia secuencial.

2 Resultados de [6]

En [6] proponemos un algoritmo que puede implementar los modelos de coherencia secuencial, causal y caché (ver la Figura 1). La selección del modelo de coherencia que implementa el algoritmo se realiza con un solo parámetro. Además, como hemos dicho, éste es el primer algoritmo propuesto que implementa coherencia caché del que tenemos noticia.

El algoritmo propuesto usa propagación y replicación. O sea, con este algoritmo cada proceso en el sistema tiene una copia del conjunto completo de las variables que consituyen la memoria compartida. Una operación de escritura se

propaga desde el proceso que la genera al resto de los procesos para que éstos la apliquen localmente en su copia de la variable correspondiente. Sin embargo, las operaciones de escritura no se propagan inmediatamente. El algoritmo se basa en un turno cíclico, con cada proceso enviando un mensaje a todos los demás en su turno. En dicho mensaje va, por cada variable, la última operación de escritura (si hay) generada por el proceso en dicha variable desde que envió el mensaje anterior. Este esquema permite controlar de una manera sencilla el número de mensajes que este algoritmo envía por la red, ya que un solo mensaje de difusión (*braodcast*) es enviado periódicamente por cada proceso. Además, en este aspecto el algoritmo se compara favorablemente con la mayoría de algoritmos que usan propagación (por ejemplo, [1,3]), ya que éstos envían un mensaje de difusión por cada operación de escritura generada, mientras que el nuestro no propaga algunas operaciones, y el resto las agrupa en un solo mensaje, con el correspondiente ahorro en capacidad de transmisión.

En nuestro algoritmo todas las operaciones son rápidas cuando implementa coherencia causal o caché. Cuando implementa coherencia secuencial sabemos por los resultados de [3] que no es posible que todas las operaciones sean rápidas. Sin embargo, todas las operaciones de escritura son rápidas, y las de lectura también lo son a menos que se produzca una condición específica en el proceso que generó la operación. Esta condición es que, desde que envió el último mensaje, haya realizado escrituras y no haya escrito en la variable a leer. Por ejemplo, no sería rápida una lectura en la variable x si el proceso que la genera, tras transmitir el último mensaje, ha generado una escritura en otra variable y y no ha generado escrituras en x. Una operación de escritura en que se cumpla esta condición debe ser bloqueada hasta que sea el turno del proceso que la generó.

El algoritmo más interesante con el que comparar nuestro algoritmo de coherencia secuencial es el algoritmo de coherencia de cachés de Afek y otros [1]. Primero, como dijimos antes, nuestro algoritmo no envía un mensaje por cada escritura en una variable como hace el suyo, por lo que en nuestro algoritmo somos capaces de controlar mejor el número de mensajes generados. Sin embargo, ambos algoritmos tienen características comunes. En ambos las escrituras son rápidas. Por lo tanto, sólo se pueden bloquear las lecturas. En su algoritmo una lectura se bloquea si hay operaciones de escritura locales que no se han aplicado en la memoria compartida. En nuestro algoritmo restringimos esta condición y sólo bloqueamos si las escrituras pendientes de ser propagadas no escriben la variable a leer. Esto hace nuestro algoritmo más interesante (aunque el algoritmo de [1] podría ser modificado para usar nuestra condición).

Otra diferencia entre ambos algoritmos es que el tiempo que una lectura puede estar bloqueada con nuestro algoritmo está acotado si los retardos de la red lo están, ya que sólo depende de este retardo y del número de procesos. En el algoritmo de [1] una lectura puede tener que esperar a que sean aplicadas un número arbitrario de escrituras. Finalmente, ambos algoritmos difieren también en el modelo usado. En el modelo de [1] se supone que hay un medio de comunicación entre los procesos que garantiza el orden total de escrituras concurrentes.

En nuestro caso no imponemos esta restricción, y logramos imponer un orden entre las operaciones con la técnica de turno cíclico descrita.

Referencias

1. Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.
2. M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, August 1995.
3. H. Attiya and J.L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.
4. A. Fekete, M.F. Kaashoek, and N. Lynch. Implementing sequentially consistent shared objects using broadcast and point–to–point communication. *Journal of the ACM*, 45(1):35–69, January 1998.
5. J.R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.
6. Ernesto Jimenez, Antonio Fernández, and Vicente Cholvi. A parametrized algorithm that implements sequential, causal, and cache memory consistency. In *Proc. of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, PDP-2002*, Gran Canaria Island, Spain, January 2002. IEEE.
7. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.
8. R. Prakash, M. Raynal, and M. Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *Journal of Parallel and Distributed Computing*, 41:190–204, 1997.
9. M. Raynal and M. Ahamad. Exploiting write semantics in implementing partially replicated causal objects. In *Proceedings of the 6th EUROMICRO Conference on Parallel and Distributed Computing*, pages 157–163, Feb 1998.

Entorno para el Diseño de Sistemas Basados en Componentes de Tiempo Real [1]

José M. Drake, Julio Luis Medina y Michael González Harbour

Grupo de Computadores y Tiempo Real.
E.T.S.I. Industriales y de Telecomunicación.
Universidad de Cantabria
Avda.de Los Castros s/n 39005 Santander - España
{ drakej, medinajl, mgh }@unican.es

Resumen. Se presenta una estrategia y un conjunto de herramientas para el diseño de sistemas de tiempo real construidos mediante ensamblado de componentes software. La metodología se basa en diseñar componentes que llevan agregado a su especificación una descripción genérica de su comportamiento temporal, de forma que cuando se construye una aplicación a partir de ellos, se construye a través de herramientas automáticas un modelo de tiempo real de la aplicación, que puede a su vez ser procesado por herramientas de diseño que ayudan a la configuración de los componentes y por herramientas de análisis que permiten garantizar su planificabilidad. El proceso de desarrollo es soportado por una herramienta CASE basada en UML que gestiona las bases de datos con las descripciones de los componentes y los modelos de las plataformas en las que se despliega la aplicación y desde las que se pueden invocar las herramientas de análisis y diseño de tiempo real.

1 Introducción

El objetivo de la tecnología de componentes software es construir aplicaciones complejas mediante ensamblado de módulos que han sido previamente diseñados por otros a fin de ser reusados en múltiples aplicaciones [1]. A la ingeniería de programación que sigue esta estrategia de diseño se la conoce por el acrónimo CBSE (Component-Based Software Engineering) y es actualmente una de las tendencias más prometedoras para incrementar la calidad del software, abreviar los tiempos de acceso al mercado y gestionar el continuo incremento de su complejidad. Aunque la tecnología de componentes es ya una realidad en muchos campos, como multimedia, interfaces gráficas, etc., plantea problemas para su aplicación en otros dominios como es el caso de los sistemas de tiempo real. Las dificultades que se presentan, no sólo surgen de la mayor diversidad y rigurosidad de los requerimientos que plantea o de las restricciones que se imponen a estos tipos de sistemas, sino principalmente de que se necesitan nuevas infraestructuras y herramientas para implementarla en los entornos de tiempo real [2,3,4,5].

[1] Este trabajo está financiado por la Comisión Interministerial de Ciencia y Tecnología mediante los proyectos TIC99-1043-C03-03 y 1FD 1997-1799 (TAP)

Sin embargo, la necesidad de la tecnología de componentes en el desarrollo de sistemas de tiempo real está siendo manifestada por la industria de automatización, eléctrica, aviación, automóvil, etc. [6,7,8,9]. Empresas líderes en estos campos como ABB han implantado la tecnología de componentes en sus productos, basándola en soluciones y especificaciones propias [6,7], lo cual les ha reportado ventajas en cuanto al mantenimiento y a la gestión de la evolución de sus productos, pero ha supuesto muchos problemas a sus clientes, que encuentran dificultades de compatibilidad de estos productos con aplicaciones desarrolladas por terceros. En otros casos [3,10,11], se ha seguido una solución diferente: se han encapsulado aplicaciones convencionales en módulos que presentan interfaces que ofrecen la conectividad y servicios propios de las tecnologías de componentes estándares mas difundidas (EJB, COM+, CCM, etc.) y se ha buscado conseguir los requerimientos propios de tiempo real, introduciendo en los componentes múltiples parámetros de configuración para que sean establecidos de acuerdo con el entorno en que vayan a operar. Esta estrategia resuelve la integración de los módulos en aplicaciones de mayor nivel a través de ensamblado, pero introduce grandes dificultades en cuanto a poder garantizar la capacidad de mantener las prestaciones no funcionales (fiabilidad, respuesta temporal, etc.) en entornos no bien definidos y que claramente no están concebidos para aplicaciones de tiempo real.

El diseño de componentes de tiempo real reusables presenta dificultades propias respecto del diseño de componentes que no son de tiempo real. En primer lugar, para conseguir las prestaciones de tiempo real se requieren mecanismos de colaboración y sincronización entre los componentes a un nivel mas bajo del que pueden ofrecer las interfaces que utiliza la tecnología de componentes como unidad de especificación e interconexión. En segundo lugar, es habitual que los sistemas de tiempo real sean sistemas embarcados, y en estos, a fin de reducir los costos y hacer posible la integración física, se limitan los recursos disponibles y se emplean entornos heterogéneos que no son los que están previstos en las tecnologías de componentes estándares. Por último, la reusabilidad de los componentes de tiempo real suele estar limitada por la necesidad de ser soportados por sistemas operativos, sistemas de comunicación o bases de datos que ofrezcan servicios específicos de gestión del tiempo, de planificación y de predecibilidad que no son habituales en los entornos de tiempo real.

El diseño de componentes de tiempo real que puedan ser ejecutados en diferentes plataformas HW/SW es una tarea compleja, ya que los componentes presentan diferente comportamiento temporal cuando se ejecutan en diferentes plataformas y en consecuencia, requieren ser verificados o reconfigurados para cada plataforma. Incluso, si se dispone de un catálogo de componentes de tiempo real bien verificados individualmente, habrá que verificar la compatibilidad entre ellos, y comprobar que los mecanismos de comunicación y sincronización retienen las prestaciones temporales previstas en la plataforma que se usa.

La línea de trabajo sobre desarrollo de componentes de tiempo real que está llevando a cabo nuestro grupo se basa en dos antecedentes ya resueltos. En primer lugar, se han desarrollado y se encuentran disponibles núcleos y sistemas operativos

de tiempo real RT_Linux y MarteOS [12,13] que ofrecen sus servicios de acuerdo con el estándar POSIX.13, lo cual proporciona un nivel de referencia estandarizada para la reusabilidad de los componentes en muchas plataformas. En segundo lugar, se han desarrollado métodos y herramientas de diseño y análisis de sistemas de tiempo real (MAST: Modeling and Analysis Suite for Real-Time Applications) [14,15,16] que permiten modelar y analizar sistemas de tiempo real basados en componentes, ya que se basan en el modelado independientemente de las plataformas y de los componentes lógicos y así mismo, ofrece las características de modularidad necesarias para generar los modelos de las aplicaciones en función de los modelos de los componentes.

2 Metodología de diseño

El aspecto central del proceso de diseño de aplicaciones basadas en componentes es la gestión de la funcionalidad a través de las especificaciones de las interfaces y esta es la misma tanto si la aplicación es de tiempo real como si no lo es. Tras una evaluación de algunos de los procesos de diseño basado en componentes descritos en la bibliografía, se ha adoptado el proceso resultante de la extensión del proceso RUP (Rational Unified Process) propuesto por Cheesman [17]. En la figura 1, se muestra un esquema a alto nivel de este proceso. Los bloques representan conjuntos de actividades que dan lugar a resultados tangibles, las flechas gruesas representan su secuenciación y las flechas finas representan el flujo de elementos generados que transfieren información entre ellas. Si comparamos este diagrama con los propuestos en la metodología RUP para la metodología orientada a objetos, se comprueba que las actividades iniciales y finales de definición de requerimientos, prueba y despliegue coinciden, mientras que difieren en que las fases centrales del proceso RUP (análisis, diseño e implementación de objetos) se han reemplazado por otras que representan la especificación, aprovisionamiento y ensamblado de componentes.

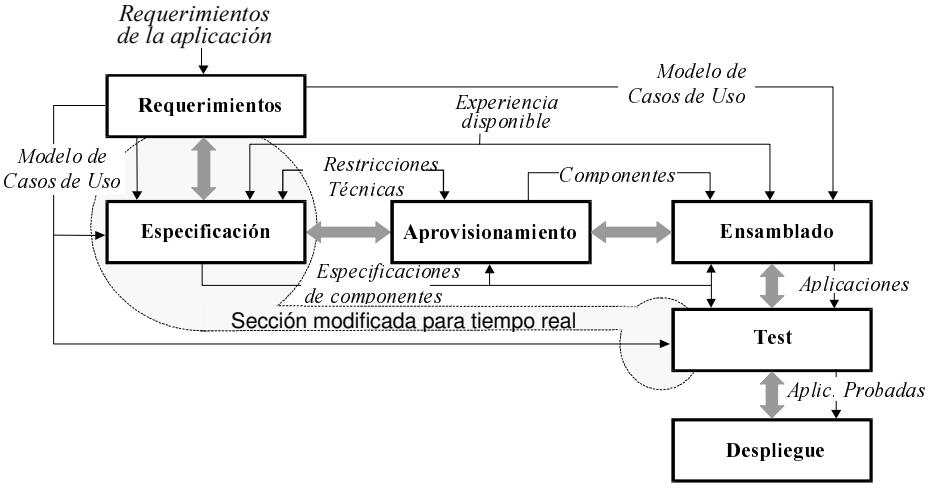


Fig. 1. "Rational Unified Process" modificado para componentes.

Los dos criterios básicos para la elección de este como base de nuestro trabajo son su carácter neutro respecto del proceso de diseño, lo que facilita las extensiones, y su formulación basada en UML que está ampliamente difundido y además lo hace compatible con las herramientas CASE que ya utilizábamos (ROSE de Rational).

La extensión a sistemas de tiempo real que hemos realizado se basa en agregar a las especificaciones de los requerimientos de las aplicaciones y a la descripción de los componentes, nuevas secciones que especifican y describen el comportamiento de tiempo real requerido y ofertado. Así, disponiendo de la descripción de cada uno de los componentes que constituyen una aplicación, junto con un modelo complementario que describe las capacidades de procesamiento de las plataformas hardware/software en que se ejecutan, se puede construir un modelo de tiempo real de la aplicación completa, que es utilizado como base de operación de las herramientas de diseño que ayudan a la configuración óptima de los componentes y de las herramientas de análisis que permiten garantizar la planificabilidad de la aplicación, o en caso negativo mediante análisis de holguras localizar las causas que le impiden satisfacer los requerimientos temporales.

La extensión que se propone solo afecta a tres de los bloques de trabajo del proceso de desarrollo de la figura 1:

- En el bloque de definición de los requerimientos, se ha realizado la extensión necesaria para que se puedan formular los requerimientos de tiempo real de la aplicación. Estos requerimientos se definen de forma independiente para cada modo de operación del sistema, y por cada uno de ellos, se formulan a través de las declaraciones de los conjuntos de transacciones que concurren en él. La declaración de cada transacción de tiempo real incluye la descripción de sus características de disparo, de los conjuntos de actividades que conlleva su ejecución y de los requerimientos de tiempo real que se imponen a lo largo de ella.
- En el bloque de actividades relativas a la especificación de los componentes, se incluyen los procedimientos de descripción del comportamiento temporal del componente. Esto se realiza mediante la especificación de la secuencia de actividades que conlleva la ejecución de cada una de los procedimientos ofrecidos por sus interfaces, y por cada actividad, se describe la cantidad procesado que requiere así como los recursos compartidos que puede requerir y que pueden dar lugar a suspensiones de su ejecución.
- Por último, en el bloque de actividades de test se incluyen las actividades que genera el modelo de tiempo real de la aplicación a partir de los modelos de los componentes que se han utilizado para su ensamblaje y de los modelos de las plataformas hardware/software en que se despliega la aplicación. Así mismo se incluyen, los procesos de análisis del modelo construido, bien para la toma de decisión sobre la configuración óptima de los componentes o bien para verificar su planificabilidad.

La idea básica de la metodología que proponemos es complementar la descripción de los componentes con un modelo de comportamiento temporal detallado, que permita modelar y analizar el comportamiento de tiempo real de las aplicaciones que

hagan uso de ellos. Nuestro método se basa en la capacidad de modelado y de análisis que tenemos, y no en la propuesta de nuevas en estrategias de diseño.

3 Modelo de tiempo real

Los conceptos y recursos de modelado de tiempo real que se proponen proceden de la metodología MAST (Modeling and Analysis Suite for Real-Time Applications) que ha sido desarrollada por nuestro grupo [14,15,16]. Su principal función es simplificar la aplicación de técnicas estándar y bien conocidas de análisis de sistemas de tiempo real, proporcionando al diseñador un conjunto de herramientas para aplicar técnicas de análisis de planificabilidad, de asignación óptima de prioridades o de cálculos de holguras, etc., sin necesidad de que tenga que conocer los detalles algorítmicos de los métodos. Actualmente, MAST cubre sistemas monoprocesadores, sistemas multiprocesadores, y sistemas distribuidos basados sobre diferentes estrategias de planificación, incluyendo planificación expulsora y no expulsora, manejadores de interrupción, servidores esporádicos, escrutinios periódicos, etc.

Características relevantes de la metodología MAST, que la hacen especialmente idónea para modelar sistemas basados en componentes, son:

- Se basa en el modelado independiente de la plataforma (procesadores, redes de comunicación, sistema operativo, drivers, etc.), de los componentes lógicos que se utilizan (requerimientos de procesado, de recursos compartidos, de otros componentes, etc.) y del propio sistema que se construye (partición, despliegue, situaciones de tiempo real, carga de trabajo y requerimientos temporales).
- El modelo se construye con elementos que modelan los aspectos de tiempo real (temporización, concurrencia, sincronización, etc.) de los elementos lógicos (componentes), y en consecuencia da lugar a un modelo que tiene la misma estructura que el código.

Permite el modelado independiente de cada componente lógico de alto nivel, a través de un modelo de tiempo real genérico, que se instancia en un modelo analizable cuando se completa con el modelo de los componentes de los que depende y se asignan valores a los parámetros definidos en el modelo.

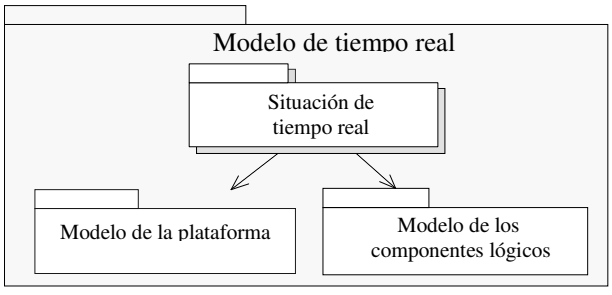


Fig. 2. Secciones del modelo de tiempo real de una aplicación.

- Soporta implícitamente la distribución de componentes, de forma que en función de que un componente se encuentre asignado o no al mismo procesador que otro del que requiere servicios, se incorpora o no el modelo de la comunicación para la invocación del servicio.

El comportamiento de tiempo real de un sistema se descompone en tres secciones independientes pero complementarias que en conjunto describen un modelo que proporciona toda la información estructural y cuantitativa que se necesita para ser procesado por las herramientas de diseño y análisis de que se dispone.

3.1 Modelo de la plataforma

Modela los recursos hardware/software que constituyen la plataforma que ejecuta la aplicación. Modela los procesadores (i.e., la capacidad de procesado, el planificador, el timer del sistema, etc.), las redes de comunicación (i.e., la capacidad de transferencia, el modo de transmisión, los planificadores de mensajes, los drivers de comunicación, etc.) y la configuración (i.e., las conexiones entre los procesadores a través de las redes de comunicación).

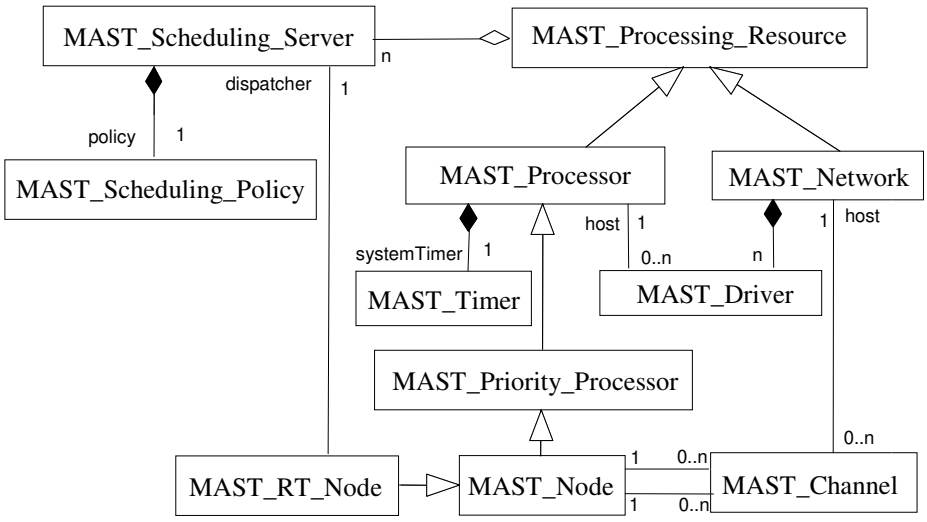


Fig. 3. Principales clases del modelo de la plataforma.

El componente básico del modelo de la plataforma es el Processing_Resource que modela un componente del sistema capaz de ejecutar actividades programadas en él, a través de conjuntos de Scheduling_Server definidos en ellos y a los que se asocian políticas especificas de planificación. En un primer nivel el Processing_Resource se especializa en Processor con capacidad de ejecutar código y Network con capacidad de transferir mensajes. Y en sucesivos niveles se definen recursos más especializados

para los que se definen atributos que describen cuantitativamente su capacidad de procesamiento o de comunicación.

El modelo de la plataforma de un sistema es un diagrama de objetos instanciados que representan los componentes que existen en la plataforma, y la interconexión entre ellos. La naturaleza de los objetos se describe a través del estereotipo, los parámetros del modelo como atributos, y sus valores concretos como valores de inicialización de los atributos.

3.2 Modelo de los componentes lógicos

Modela el comportamiento de tiempo real de los componentes software con los que se construye la aplicación. El modelo de un componente software describe:
- La cantidad de procesado que requiere la ejecución de las operaciones de su interfaz.
- Los componentes lógicos de los que requiere servicios.
- Los procesos o threads que crea para ejecutar sus operaciones.
- Los mecanismos de sincronización que requieren sus operaciones.
- Los parámetros de planificación definidos explícitamente en el código.
- Los estados internos relevantes a efecto de requerimientos de tiempo real.

En la figura 4 se muestran las clases de mas alto nivel que se utilizan para formular el modelo de los módulos software y en particular de los componentes.

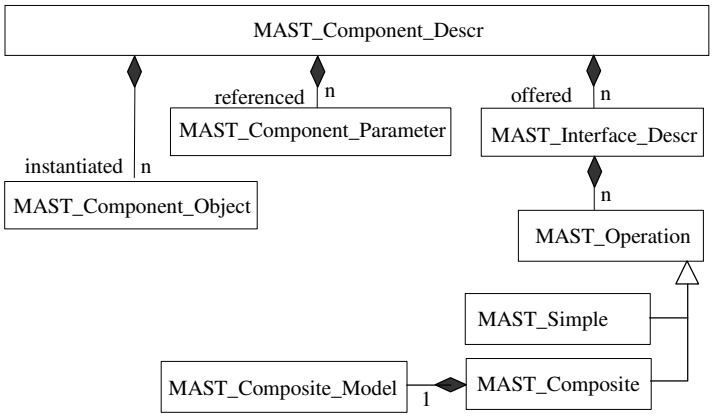


Fig. 4. Principales clases del modelo de tiempo real de los componentes lógicos.

La clase raíz es MAST_Component_Descr. Las instancias de esta clase describen el modelo de un componente. Es básicamente un elemento contenedor de:
- Modelos de operaciones (organizados por interfaces).
- Componentes agregados a él, que serán instanciados por cada instancia del componente.

- Parámetros con los que se configura cada instancia para que represente un modelo específico.

El elemento básico que exporta un MAST_Component es el modelo de temporización de las operaciones que ofrece la interfaz del componente lógico. A diferencia de la descripción lógica, una interfaz no tiene un modelo de tiempo real propio, sino que únicamente constituye un elemento contenedor en el que se organizan los modelos de las operaciones dentro del componente.

Los parámetros declarados en la declaración de un componente constituyen la clave de la capacidad de modelar un tipo de componente mediante un único objeto, que posteriormente a través de sus múltiples instancias sea capaz de describir el comportamiento temporal de componentes bajo entornos muy diferentes. Los parámetros representan diferentes tipos de elementos que son utilizados para describir los modelos de las operaciones. En la descripción del componente los parámetros son solo referencias indefinidas. Sin embargo, en la instanciación del componente le son asignados valores concretos que hacen referencia a objetos instanciados y que completan el modelo de la instancia, y hace posible que diferentes instancias de un mismo componente puedan modelar comportamientos diferentes. Los tipos de elementos que pueden ser referenciados en un componente a través de parámetros, son:

- MAST_Component_Object: Otros componentes de los que depende su funcionalidad.
- MAST_Scheduling_Server: Servidores de planificación que soporten la concurrencia interna de sus actividades.
- MAST_Shared_Resource: Recursos compartidos que son accedidos con exclusión mutua.
- MAST_Operation: Modelo de operaciones.
- MAST_Timing_Requirement: Requerimientos temporales asociados a estados internos
- MAST_External_Event_Source: Modelos de generación de eventos.
- MAST_External_State: Estados internos relevantes de las operaciones.

Los objetos de la clase MAST_Operation modelan la cantidad de procesado y los requerimientos de sincronización de la ejecución de los procedimientos definidos en la interfaz del componente. A alto nivel existen dos clases de operaciones:

- MAST_Simple: Modela la ejecución de una sección de código que no hace referencia a ninguna otra operación. Describe la duración de su ejecución a través de tres parámetros predefinidos: “wcet” worst case execution time, “acet” average case execution time y “bcet” best case execution time. Estos parámetros se describen en tiempo normalizado, que es transformado en tiempo físico cuando se divide por el “Speed_Factor” del recurso de procesamiento en que se ejecuta.
- MAST_Composite: Modela la ejecución de una sección de código que hace referencia (depende) de otras operaciones definidas en la interfaz del mismo componente o de otros componentes. Una operación MAST_Composite requiere ser declarada y también ser descrita. Esto se realiza mediante un diagrama de actividad que se agrega a la propia declaración y que describe las secuencias de actividades que conlleva su ejecución. Así mismo, puede tener definidos

parámetros, lo que particulariza cada invocación de acuerdo con los valores que se le asignan.

3.3 Modelo de las situaciones de tiempo real

Una "Situación de Tiempo Real" representa un modo de operación del sistema junto con un modelo de la carga de trabajo que tiene que ejecutar. Constituye el ámbito en que operará una herramienta de diseño o de análisis. Aunque los modelos de las situaciones de tiempo real son planteados independientemente, comparten el modelo de la plataforma y el modelo de muchos de los componentes software que se utilizan en ellas.

En la figura 5 se muestra los componentes básicos que constituyen el modelo de una situación de tiempo real. Los dos aspectos básicos que describen a una situación de tiempo real son el conjunto de componentes software que se instancian para su ejecución y el conjunto de transacciones que pueden concurrir en ella.

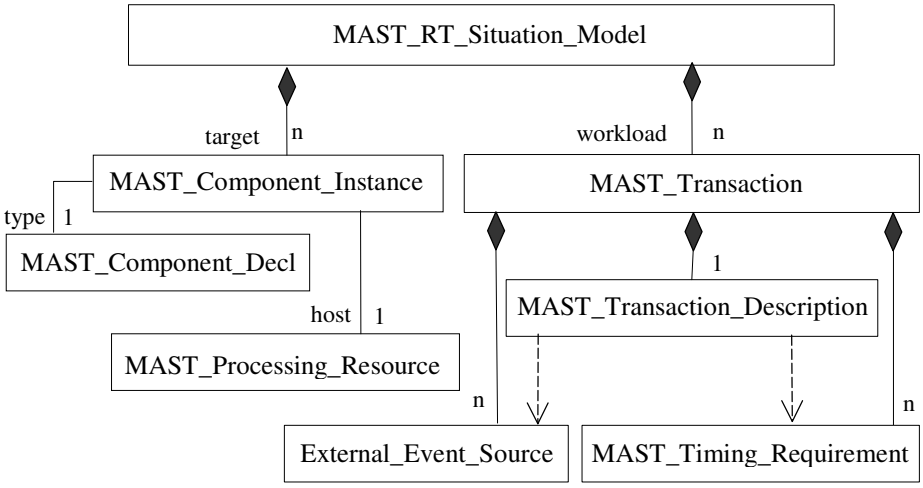


Fig. 5. Principales clases para la descripción de una situación de tiempo real.

En el modelo de una situación de tiempo real se describe la configuración de la aplicación, que consiste en la declaración de cada una de las instancias de componentes software que participan en alguna de las actividades de las transacciones de la situación. La instancia de un componente implica a su vez la instanciación recursiva de todos los componentes que estaban declarados en él. Así mismo, al asignar un identificador a la instancia, se proporciona recursivamente un identificador absoluto (aunque compuesto) a cada componente agregado.

Aunque los objetos de la situación que se modela pueden instanciarse dinámicamente dentro de ella, el modelo de tiempo real es siempre estático, esto es,

todos los objetos que participen han de estar declarados a un mismo nivel y tener asignado un identificador estático.

La configuración de la situación implica también la declaración de su despliegue sobre la plataforma, y a tal fin, a cada instancia de componente software que se declara, se le asigna el procesador en que se ejecuta. Todos los componentes que se instancian por agregación a partir de un componente software, están asignados al procesador a que se asignó este.

La carga de trabajo de una situación de tiempo real se modela como un conjunto de transacciones. Cada Transacción describe una secuencia no iterativa de actividades que se desencadenan como respuesta a un patrón de eventos externos (trigger) que a su vez, sirve de referencia para definir los requerimientos temporales. Cada transacción incluye todas las actividades que se desencadenan como consecuencia del evento de entrada y todas aquellas que requieren sincronización directa con ellas (intercambio de eventos, sincronización por invocación, etc.). No necesitan modelarse dentro de una transacción, otras actividades concurrentes que influyen en su evolución a través de competir por el acceso a recursos comunes, ya sean recursos de procesamiento (Processing_Resource) u objetos protegidos a los que se accede con exclusión mutua. La presencia de estos recursos que son compartidos por las actividades de varias de las transacciones que coexisten concurrentemente dentro de una misma situación de tiempo real y que implican bloqueos en la ejecución de sus actividades, es la causa de que el análisis de tiempo real deba hacerse contemplando simultáneamente todas las transacciones que concurren en la situación de tiempo real.

Las transacciones de tiempo real son componentes básicos de la concepción y especificación de un sistema de tiempo real. Sin embargo, tienen su origen en diferentes fases de su proceso de diseño:

- Transacciones que resultan de la formulación de los requerimientos temporales de las aplicaciones (Event Driven) que se plantean como secuencias de actividades que se desencadenan como respuesta a un evento externo de trigger, junto con las restricciones temporales que se imponen relativas a los instantes en que deben haber concluido estas actividades respecto del trigger que la originó.
- Transacciones que resultan de requerimientos de tiempo real impuestos en la fase de diseño del sistema, como mecanismo interno para satisfacer requerimientos del sistema que en origen pueden no ser de tiempo real.
- Transacciones que se introducen como artificios de análisis para garantizar los requerimientos temporales bajo condiciones de peor caso de sistemas sin eventos externos y con requerimientos temporales entre eventos internos.
- Transacciones sin requerimientos de tiempo real pero que afectan a transacciones con requerimientos de tiempo real que concurren con ellas.

Aunque los cuatro tipos de transacciones tienen origen diferente durante el diseño, convergen al implementarse todas ellas como respuestas a eventos externos generados bien por dispositivos I/O o por el hardware de temporización.

La declaración de una transacción requiere la declaración de tres componentes:

- Trigger_Pattern: Describe la distribución temporal en la generación de la secuencia de eventos externo que la dispara.
- Transaction_Activity: Describe la actividad que se desencadena por cada evento que se produce. Esta actividad se describe mediante un diagrama de actividad agregado a la declaración de la transacción.
- Lista de Timing_Requirements: En cada transacción se definen el conjunto de requerimientos temporales que deben ser satisfechos durante su ejecución.

4 Herramienta de análisis y diseño de tiempo real.

El proceso de desarrollo de aplicaciones basado en componentes que se citó en el apartado 2, requiere de una herramienta que lo soporte. La descripción de cada componente con sus dos vistas, la del que lo usa y la del que lo instala, requiere una información muy extensa que debe ser cotejada para validar la compatibilidad de conexión de los componentes entre sí. En el caso de diseño de sistemas de tiempo real, la herramienta se sitúa en el centro del proceso, ya que el modelo de tiempo real de la aplicación tiene que ser construido a partir de la información de los componentes, y su explotación para las fases de diseño y análisis requiere de la potencia de cálculo para llevarlo a cabo.

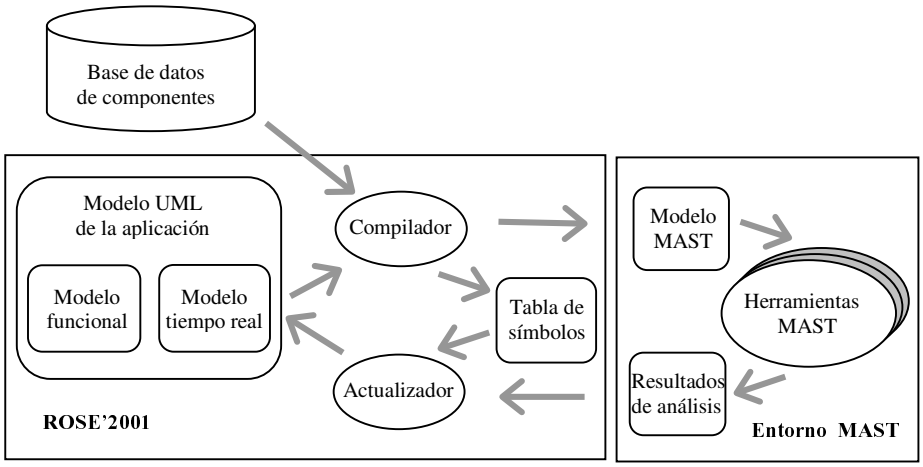


Fig. 6. Conjunto de herramientas para diseño de tiempo real.

Existen tres elementos básicos que constituyen el entorno de instrumentación:

- La herramienta que soporta el proceso de diseño de la aplicación. Está construida a partir de la herramienta ROSE de Rational y su función es servir de soporte de las estructuras UML con las que el diseñador define la aplicación, y las herramientas específicas que generan los modelos de tiempo real y gestionan el acceso a las herramientas externas.

- Las bases de datos de los componentes que contiene las descripciones funcionales y los modelos de tiempo real de los componentes en que se basa el diseño. Estas bases de datos se construyen actualmente con estructuras de ficheros que son a su vez compatibles con los formatos de ROSE.
- Las herramientas de diseño y análisis de tiempo real del entorno MAST, que utilizan ficheros de entrada y salida con formato propio. Entre estas, las que se tienen actualmente desarrolladas (✓) o en desarrollo (-) son:
  - ✓Holistic analysis
  - ✓Offset-based analysis
  - - Varying priorities analysis
  - - Multiple event analysis
  - ✓Monoprocessor priority assignment
  - ✓Linear HOPA (Holistic optimum priority assignment)
  - ✓Linear simulated annealing priority assignment
  - - Multiple event priority assignment
  - - Monoprocessor simulation
  - - Distributed simulation

El entorno de diseño que se ha descrito en esta sección es aún parte de un proyecto que se está desarrollando. Muchos de sus componentes, como son la herramienta que soporta la aplicación y las herramientas que implementan los algoritmos de diseño y análisis de tiempo real están ya construidas, pero otros como las bases de datos de los componentes, y los compiladores para la construcción de los modelos de la aplicación aún están en desarrollo.

5 Conclusiones.

Se está realizando la extensión de un entorno de desarrollo de sistemas basados en componentes para que soporte el diseño de sistemas de tiempo real. Esta extensión se basa en la capacidad de modelado del comportamiento temporal de los servicios que ofrecen los componentes y en la generación automática a través de herramientas CASE de modelos de la aplicación, que son procesados a su vez por herramientas de diseño y de análisis de tiempo real.

Con el uso de este entorno, el diseñador puede construir la aplicación por ensamblado de componentes ya elaborados, aunque posiblemente tenga también que diseñar y ensamblar alguno específico que sea propio de la aplicación. Esta fase de diseño está en gran medida conducida por la funcionalidad que se requiere en la aplicación y por la funcionalidad que ofrecen los componentes disponibles. Obviamente si existen requerimientos de tiempo real, en esta fase deben ser también consideradas las capacidades de respuesta temporal de los componentes y seguir en el diseño patrones software que sean adecuados para la planificabilidad de la aplicación. A partir de este punto es cuando se incorporan las capacidades de tiempo real que se han introducido con nuestra extensión. En primer lugar, la herramienta va a proporcionar un modelo global de la aplicación que se genera a partir de los modelos

de comportamiento de los componentes y del modelo de las plataformas hardware/software en que se despliega. En segundo lugar, las herramientas de diseño disponibles ayudan al diseñador en la configuración de la aplicación. Así se puede obtener información sobre las prioridades óptimas de los threads en los procesadores o de los mensajes en las redes, el diseño adecuado de concurrencia y la asignación adecuada de actividades a threads, la asignación adecuada de componentes a los nudos de la plataforma, etc. Por último, las herramientas de análisis podrán garantizar la planificabilidad de la aplicación, si lo es, o identificar los elementos del software que lo impiden.

La principal dificultad que encontramos, para el desarrollo de esta metodología de tiempo real, es disponer de entornos de ejecución de componentes estándares que tengan prestaciones de tiempo real estricto y que sean compatibles con las configuraciones mínimas que son propias de los sistemas embarcados. Las validaciones de la metodología que por ahora hacemos se basan en componentes encapsulados como paquetes Ada y que se ejecutan en plataformas dotadas con sistemas operativos de tiempo real contrastados.

Referencias

1. Szyperski C.: "Component Software: Beyond Object-Oriented Programming" Addison-Wesley, 1999.
2. Isovic D. y Norström C.: "Components in Real-Time Systems". The 8th Int. Conf. On Real-Time Computing Systems and Applications (RTCSA'2002), Tokyo (Japan), 2002.
3. Yau S.S. y Xia B.: "An approach to Distributed Component-based Real-time Application Software Development" Proc. IEEE Int´l Symp.Object-oriented Real-Time Distributed Computing (ISORT'98), April, 1998, pp. 275-283.
4. Lehman M.M. y Ramil J.F.: EPiCS: Evolution Phenomenology in Component-intensive Software" Proposal draft, Dept. of Computing of Imperial College, London, Aug. 2001.
5. Welling A. and Cornwell, P.: "Transaction Integration for reusable hard Real-time Components" Proc. of Ada in Europe, pp. 365 - 378, Springer-Verlag April, 1996.
6. Crnkovic I. Y otros: "A case Study: Demands on Component-based Development" Proc. Of 22nd Int. Conf. Of Software Engineering, Cannes (France), May 2000.
7. Larsson M. y otros: "Development Experiences of a Component-based System" IEEE Proc. of Engineering of Computer Based Systems (ECBS-2000), 2000
8. Norström C. y otros: "Experiences from Introducing State-of-the-art real-time Techniques in the Automotive Industry". Proc. Of 8th IEEE Int. Conf. On Engineering of Computer Based Systems (ECBS01) Washington, April, 2001.
9. Sha L. Rajkumar R. and Gagliardi M.: "Dependable System Upgrade" IEEE Proc. 19th Real-Time Systems Symposium, pp. 440-448, Madrid 1998.
10. Cook J.E. y Dage J.A.: "Highly Reliable Upgrading of Components" Proc. 21st Int. Conf. On Software Engineering. Los Angeles (USA), 1999.
11. Chang E., Annal D. y Grunta F.: "A Large Scale Distributed Object Architecture CORBA&COM for Real Time Systems" IEEE database, 0-7695-0607-0/00., 2000.
12. Aldea M. y González Harbour M.: "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications." International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, LNCS, May 2001.

13. Aldea M. y González Harbour M.: "POSIX-Compatible Application-Defined Scheduling in MaRTE OS". Proceedings of 13th Euromicro Conference on Real-Time Systems (WiP), Delft, The Netherlands, June 2001.
14. González Harbour M., Gutiérrez J.J, Palencia J.C. and Drake J.M.: "MAST: Modeling and Analysis Suite for Real-Time Applications" Proceedings of the Euromicro Conference on Real-Time Systems, June 2001.
15. Medina J.L.,González Harbour M.,. Drake J.M:" "MAST Real-time View: A Graphic UML Tool for Modeling Object_Oriented Real_Time Systems", RTSS, 2001, December, 2001.
16. Medina J.L., Gutierrez J.J., González Harbour M., Drake J.M.: "Modeling and Schedulability Analysis of Hard Real-Time Distributed Systems based on ADA Componentes." Ada Europe, 2002
17. Cheesman J. and Daniels J.: "UML Components: A simple Process for Specifying Component-Based Software", Addison-Wesley,2000.
18. D'Souza D.F. and Wills A.C.: "Objects, Components and Frame-works in UML: the Catalysis approach". Addison-Wesley, 1998.
19. Gomaa H.: "Designing Concurrent, Distributed and Real-Time Application with UML". Addison-Wesley, 2000.

Sistema de instrumentación y control distribuido BRAVO 2000

Fernando Quero Sanz[1], Javier Cuevas Domingo[2], David Asiain Ansorena[3], J. Luis Vela Pérez[4]

Escuela Universitaria Politécnica La Almunia, c/ Mayor S/N, 50100 Zaragoza, Spain

1 fernando@eupla.unizar.es, 2 javier.cuevas@eupla.unizar.es,
3 david.asiain@eupla.unizar.es, 4 joseluis.vela@eupla.unizar.es

Abstract. Este documento pretende reflejar las principales ventajas que aporta un sistema de instrumentación y control distribuido, así como la justificación de su diseño en aspectos tales como; tecnología bus CAN, topología de red y protocolo de comunicación.

1 Introducción

El objeto de este proyecto supone una alternativa en cuanto a la estructura típica en sistemas de adquisición de datos y control. Existen dos formas de enfocar este tipo de sistemas, por un lado la centralización de funciones en una determinada arquitectura y por otro una distribución de funciones en diversos módulos o arquitecturas más simples.

Sistemas Centralizados. En estos sistemas existe un procesador central que se encarga de realizar todas las funciones del sistema, siendo necesaria una plataforma de altas prestaciones con un rendimiento suficiente para atender a todas las necesidades del sistema en cuanto a tiempo de respuesta y potencia de procesado.

Sistemas Distribuidos. En estos sistemas, por el contrario, se produce una descentralización de funciones en módulos de carácter especifico. Todos estos módulos conformarán una red, siendo necesario establecer un protocolo estándar de comunicación. A uno de estos módulos se le podría asignar una funcionalidad de tipo cliente, el cual contendría un software de interface con la red para la realización de las típicas tareas de configuración, control y adquisición de información.

Los sistemas más difundidos en la última década son una mezcla entre ambas topologías. Estos consisten frecuentemente en redes de PLC's conectadas a un procesador central, normalmente un PC, sobre el que corre una aplicación de tipo SCADA, que permite interactuar sobre todo el entorno.

Efectivamente se produce una distribución de funciones sobre cada autómata conectado, conformando redes gestionadas por protocolos estándar tan difundidos como ProfiBus, FieldBus, ModBus, etc. Sin embargo y debido a la potencia que conlleva la incorporación de un autómata, éste puede llegar a ser el procesador maestro de un entorno local con topología centralizada.

2 Ventajas de los sistemas distribuidos

Las ventajas de un sistema distribuido o arquitectura en red, son notables frente a un sistema centralizado. La idea básica es dividir un problema de características y topología compleja, en una serie de unidades o módulos más sencillos de aplicación específica.

Cada módulo funcional incorpora un μC (microcontrolador) que controla el hardware necesario para desarrollar una determinada tarea. En función de la tarea específica, el módulo resultante posee una complejidad variable, y por tanto se pueden diseñar desde simples módulos con puertos digitales I/O optoacoplados hasta complejos módulos de adquisición de datos de alta velocidad con procesado de señal en el dominio de la frecuencia.

Independientemente de la tarea que realicen los módulos, los sistemas distribuidos requieren de un sistema de comunicación o bus. Todos los módulos son conectados entre sí a través de este bus, permitiendo el control y la configuración de cada unidad, además de permitir el intercambio de datos entre los módulos de la red.

Se ha pasado de tener un sistema centralizado que realiza todas las tareas de control y adquisición de datos, a delegar estas tareas en módulos de funcionalidad específica capaces de comunicarse a través de un bus. Esta descentralización de tareas y por tanto la especialización de los módulos en el problema especifico a resolver permite dotar a los módulos de un cierto grado de inteligencia que redundará en un incremento notable en cuanto a la detección y solución de averías y problemas.

Además, este tipo de arquitectura, hace más simple la instalación, es más flexible, y normalmente facilita la introducción de cambios en una estructura ya implementada.

Hay que destacar que al delegar en los módulos la mayoría del trabajo, los sistemas distribuidos, están facilitando la incorporación de sistemas de control basados en tecnología PC aportando todas las facilidades tanto de desarrollo de aplicaciones como de facilidad de uso, que han hecho tan popular a este entorno de trabajo, sin perder robustez ni rendimiento.

Las mejoras en escalabilidad, diagnóstico de errores, mantenimiento, simplificación de las instalaciones, su fácil integración con otros sistemas existentes junto con el

continuo abaratamiento del diseño de sistemas digitales, están haciendo inclinar la balanza hacia los sistemas distribuidos frente a los sistemas tradicionales.

La mayor directiva en la industria del control y adquisición de datos es la máxima de "fácil de usar". Esto es debido a que "fácil de usar" da soporte al desarrollo de aplicaciones libres de errores y de una forma rápida; además de facilitar el mantenimiento, lo cual revierte sin duda en un ahorro considerable de dinero. Las compañías de control han empezado a analizar como pueden facilitar el trabajo de los integradores de sistemas. Una clave en este tipo de estrategias, hoy en día es la integración de sistemas orientados al PC. La tecnología PC promete ser una solución abierta y muy flexible.

La principal característica de los PC's es su habilidad para realizar diferentes trabajos. El PC puede realizar las funciones de un controlador de celdas, un supervisor de estaciones, un controlador lógico, etc. La flexibilidad de la arquitectura PC permite ejecutar todas estas tareas diferentes en la misma plataforma hardware. Los requerimientos están evolucionando hacia un enlace de comunicación universal en lugar del uso de redes especializadas y difíciles de usar.

Basados en la idea de "fácil de usar", surge la necesidad adicional de añadir requerimientos de diagnostico y configuración al nivel de dispositivos. Los diagnósticos a este nivel, permiten a los usuarios identificar específicamente un dispositivo problemático e incluso identificar el tipo de problema gracias a la especialización del módulo La configuración de dispositivos es necesaria para ofrecer un amplio rango de productos que pueden ser configurados para satisfacer las necesidades de procesos específicos de los clientes. Los dispositivos configurables optimizan los procesos y reducen la complejidad del mantenimiento reduciendo el riesgo de seleccionar el dispositivo incorrecto cuando se realizan sustituciones.

Tomando como base estas directrices, se ha realizado el diseño de un sistema universal de instrumentación y control, como una solución que integra la tecnología BUS-CAN orientada al dispositivo y módulos I/O inteligentes con un potente software sobre arquitectura PC.

3 Sistema universal de instrumentación y control BRAVO 2000

BRAVO 2000 es un sistema distribuido de instrumentación, modular y extremadamente versátil, que permite su adaptación a cualquier ambiente y situación en el que se tenga que monitorizar algún tipo de proceso, desde por ejemplo, un ensayo destructivo en laboratorio, hasta un ensayo de un vehículo en movimiento. Gracias a su versatilidad, fácil mantenimiento y modularidad también puede adaptarse a la industria, instrumentando líneas de fabricación para mejorar el control y la automatización de procesos, con un reducido mantenimiento y bajo coste de la

instalación al establecerse, una sola manguera de cuatro hilos para BUS-CAN y alimentación respectivamente.

El sistema satisface las funciones de acondicionamiento, adquisición y preprocesado de las señales de los sensores; adicionalmente puede estar encargado de la gestión de hipótesis en un ensayo, gestión de históricos en una línea de producción, adquisición desde sensores inteligentes y actuadores remotos, así como de la comunicación con otros sistemas como PLC´s. Merece ser destacado su alto nivel de conectividad, pues puede conectarse con sistemas RS232, RS485 y RS422; es factible establecer conexión remota vía radio e incluso se puede tener control de la aplicación a través de Internet gracias a la tecnología cliente-servidor sobre TCP/IP.

Otro aspecto importante es la posibilidad de enlazar diversos segmentos CAN del sistema a través de un módulo router CAN-Ethernet. La incorporación de un módulo cliente personalizará cada aplicación en cuestión; existe un módulo genérico que permite realizar las tareas típicas de análisis de datos en el dominio del tiempo y de la frecuencia. La topología del sistema evita la necesidad de utilizar múltiples aplicaciones de distintos fabricantes e incompatibles entre sí.

3.1 Experiencia previa

El sistema BRAVO 2000 es fruto de la experiencia del Centro de I+D de la EUPLA en la instrumentación de ensayos experimentales y monitorización de procesos. Desde hace más de diez años este Centro ha ido adquiriendo experiencia en la instrumentación de ensayos de campo y de laboratorio con las últimas tecnologías en sensores y materiales.

Se ha trabajado en diversos tipos de ensayos experimentales, desde ensayos simples de probetas en laboratorio, a ensayos de campo, con más de un centenar de sensores distribuidos sobre el espécimen bajo estudio.

Entre otros destacan la instrumentación completa y desarrollo del sistema de navegación del barco de vela que representó a España en la Copa América de 1999 en Auckland (Nueva Zelanda). Competición en la que la tecnología cobra un papel determinante. Aquí se tuvo la oportunidad de trabajar con sensores de última tecnología para obtener un modelo inercial y de posicionamiento en los tres ejes, además de instrumentar la mayoría de partes del barco sometidas a esfuerzos, incluso en zonas sumergidas, para con todos estos datos realizar rigurosos análisis enfocados a aligerar los diversos elementos que configuraban el barco y así aumentar al máximo su rendimiento.

Más que de un barco se trataba de un banco de ensayos en las peores condiciones de salinidad y humedad, donde el bajo consumo, la facilidad de mantenimiento y de

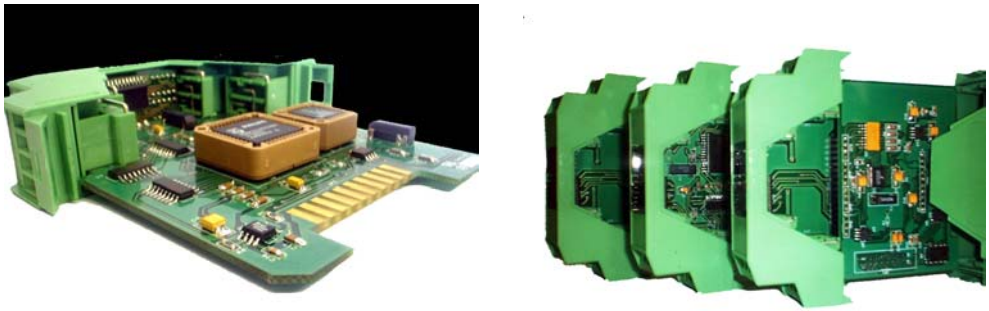
instalación fueron determinantes para asegurar el buen funcionamiento del sistema. Así mismo tenía gran importancia la fácil adaptación de todo el sistema, con respecto a las diferentes hipótesis de los ensayos a los que se sometía el barco en el día a día; tanto en instalación como en software de control

Otro apartado importante de destacar de la mano de EADS CASA Espacio son los ensayos realizados sobre diversos elementos que configuran la estructura de Ariane 5, así como ensayos específicos sobre satélites y despliegue de antenas. Por otro lado se ha realizado con SENER Ingeniería y Sistemas, ensayos de seguridad pasiva para el sector de la automoción, estudiando el efecto del impacto de articulaciones humanas sobre elementos tales como; parachoques, salpicaderos, etc.

La trayectoria profesional de la EUPLA en este ámbito, queda reflejada en un sistema distribuido de adquisición y control, extremadamente versátil, de bajo consumo, fácil mantenimiento y controlado por un potente software.

3.2 Sistema distribuido modular

El sistema consta de una serie de módulos independientes entre sí y conectados mediante un solo cable con tecnología BUS-CAN. Estos módulos son interfaces independientes y específicos en el tratamiento de la señal con respecto al tipo de sensor o actuador que manejan.



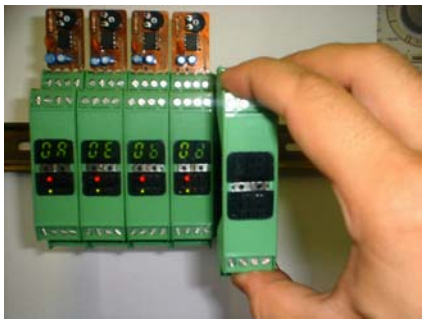
Así mismo existen módulos de comunicaciones que establecen diferentes tipos de interfaces, tales como; RS232, RS485, RS422, RF/BUS-CAN y Gateaways. De esta forma el sistema adquiere un alto grado de conectividad que le permite comunicarse y controlar la mayoría de sistemas industriales como PLC´s o sensores inteligentes que rutan información ya preprocesada a través de protocolos serie. Además permite establecer redes de sensores o actuadores remotos gracias a su interface RF CAN con el que es factible monitorizar ensayos de campo tales como, la instrumentación y ensayo de un vehículo en circuito.

3.3 Fácil instalación y mantenimiento

La red de módulos que forman este sistema esta unida por un solo cable de dos pares de hilos. Uno de ellos dedicado a alimentación y otro dedicado a transferencia de información mediante BUS-CAN. De esta forma, son los propios módulos los que alimentan a los sensores conectados, reduciéndose de forma drástica el número de cables, con el consiguiente ahorro de coste en la instalación y disminución del mantenimiento necesario a lo largo de su vida operativa, puesto que es mucho más sencillo mantener una instalación con un número reducido de cables.

Además el protocolo de comunicaciones EUPLA-CAN. Desarrollado por el I+D de la EUPLA, permite la auto-detección de fallos en toda la subred de módulos, tanto en el ámbito de módulos como en el ámbito de sensores y actuadores conectados.

Estos módulos se han desarrollado con tecnología HARD PLUG&PLAY, es decir, se pueden añadir o quitar módulos en la red sin necesidad de desconectar la alimentación del sistema y sin necesidad de hacer un reset sobre el mismo. Esta característica es fundamental en líneas de producción en las que las interrupciones suponen grandes perdidas, o en ensayos de larga duración y elevado presupuesto.



3.4 Sencillez, bajo coste y bajo consumo.

Los módulos que integran el sistema se basan en la sencillez, resolviendo problemas muy concretos como el acondicionamiento de un sensor determinado. Gracias a esta

sencillez, el coste de una instalación puede quedar considerablemente reducido, ya que la adaptación del sistema a una aplicación específica, permite dimensionarla en una mayor proporción, no redundando en la incorporación de equipos con funcionalidades que jamás serán usadas.

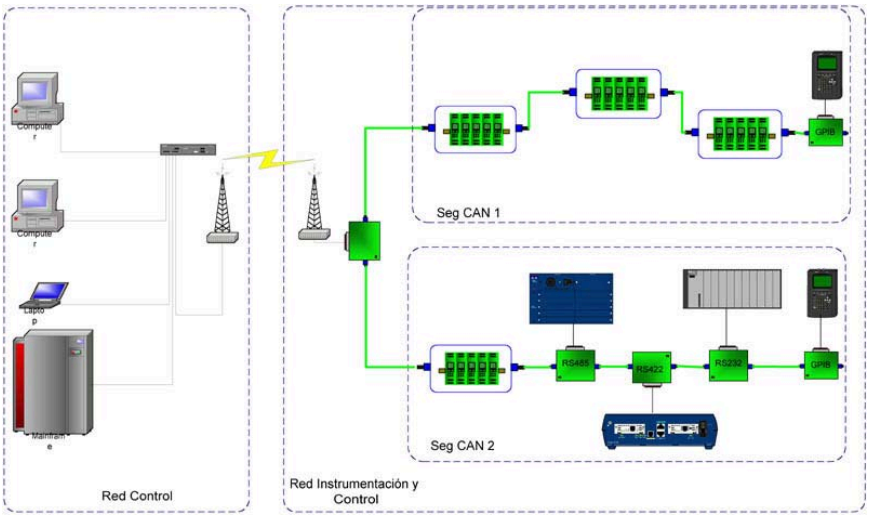
Además los módulos se desarrollan con tecnología SMD que permite disminuir al máximo su tamaño y consigue unos consumos verdaderamente reducidos. Esta característica es fundamental en ensayos de campo, en los que la alimentación se realiza mediante baterías; como sería el caso de la instrumentación de un velero, una motocicleta o automóvil en un circuito de pruebas. En este caso son de vital importancia estas características de bajo consumo, reducido espacio, y sencillez de la instalación a través de un solo cable.

3.5 Topología de Red

3.5.1 Introducción

Los diversos módulos que integran el sistema están conectados a través del bus a un Host o módulo principal que se encarga de controlarlos. Adicionalmente este módulo se encarga de servir datos a la aplicación cliente, encargada de mostrar datos y de ejercer de interface con el usuario.

El módulo principal o servidor de datos, es un sistema empotrado encargado de controlar la adquisición de datos y de asegurar su robustez. Este módulo puede funcionar sin la aplicación cliente de interface, y asumir todo el trabajo de la configuración de la red y control de la adquisición. En cualquier caso, la configuración de este módulo principal se establecerá desde el interface de la aplicación del cliente. Además el módulo principal puede soportar múltiples segmentos de red con lo que es factible conseguir un elevado número de puntos de adquisición y control.



La distribución de módulos puede adaptarse a la aplicación concreta en la que el sistema sea adaptado. Los sensores o actuadores se cablean a los módulos y estos a su vez se conectan entre sí y al módulo principal a través de un solo cable en bus CAN que permite una distancia de bus de hasta 10 Km. Por su parte el módulo principal es conectado a la aplicación cliente a través de Ethernet. La distribución de los módulos puede ser tanto cercana a los sensores y actuadores, con la correspondiente reducción del cableado, o bien distribuyendo los módulos en unos armarios o racks para su centralización, de forma que se consiga agrupar los módulos de una misma zona; conectando dichos racks entre sí y con módulo principal a través del bus.

Por lo tanto se observa que la topología de la red es muy flexible y se adapta a cualquier morfología de ensayo o proceso a instrumentar.

3.5.2 Transporte de tramas CAN sobre TCP/IP

De la misma forma la aplicación cliente puede soportar múltiples módulos principales a través de LAN, la conexión entre dos módulos que estando en un segmento diferente tuvieran que intercambiarse una serie de tramas CAN, queda resuelto con el diseño de un protocolo de transporte de tramas CAN sobre TCP/IP, de esta forma es factible conseguir dar cobertura en una instalación de elevadas dimensiones, aprovechando la propia instalación de cableado LAN.

El diseño del protocolo se ha realizado teniendo en cuenta los siguientes aspectos:

- El protocolo no debe hacer en ningún caso, referencia al contenido de las tramas CAN transportadas.
- La comunicación entre los dos nodos debe de ser bidireccional.
- Deben ser proporcionados mecanismos que permitan negociar las condiciones en que se va a dar la comunicación antes del envío de alguna trama CAN.

Partiendo de estas premisas desde el punto de vista del establecimiento de la conexión pueden observarse dos entidades:

El Servidor: Es el encargado de la aceptación de conexiones. Deberá de poder establecer las condiciones bajo las cuales se acepte la conexión o se rechace. A éste proceso se le denominará validación de la conexión. Una vez satisfecha la validación por parte del cliente, tanto el cliente como el servidor tendrán derecho a solicitar al otro, la existencia o no de características de modos de comunicación. A este proceso se le denominará “negociación del enlace”. La “negociación del enlace” se dará por concluida cuando el cliente de por acabada la negociación.

El Cliente: Es aquél que solicita la conexión. Deberá de satisfacer las condiciones de validación que solicite el servidor. Posteriormente tendrá la obligación de contestar a

las solicitudes del servidor de la existencia o no de características de modos de comunicación. Por último dará por finalizada la negociación, estableciendo el enlace.

Una vez establecida la conexión se considera que los datos podrán fluir en ambas direcciones por igual. No existe un maestro ni un esclavo. Cualquiera de las dos entidades puede terminar la conexión. Se debe, en la medida de lo posible, aprovechar al máximo el ancho de banda del nuevo medio.

En este intercambio de paquetes CAN debe de asegurarse dos aspectos fundamentales:

- Las tramas CAN deben estar libres de fallos, es decir las tramas no han sufrido modificaciones en el transporte.
- El orden de las tramas es consistente tanto desde el punto de vista del emisor como del receptor. Se espera recibir las tramas en el mismo orden en el que se han emitido.

Los principales objetivos que debe de satisfacer el diseño del protocolo serán:

1. Diseñar la forma en la que se empaquetarán las tramas CAN para transmitirlas vía TCP/IP.
2. Establecer una conexión entre el cliente y el servidor:
   - Validar el proceso cliente. (Validación)
   - Establecer las pautas de la posterior comunicación. (Negociación)
3. Una vez establecida la conexión:
   - Poder enviar datos en ambos sentidos por igual.
   - Usar óptimamente el ancho de banda.
   - Respetar el orden de llegada de los paquetes.
   - Poder validar que los datos recibidos son iguales que los enviados (errores).

3.5.3 Segmentación de la red CAN mediante Gateway

Puede ser interesante en ocasiones realizar una segmentación de la red CAN, con objeto que la distancia entre módulos y la velocidad de comunicación (Baudrate) no sea óbice para alterar el rendimiento general de la red. Para conseguir este objetivo se hace necesario el diseño de un nuevo módulo denominado Gateaway, cuyo propósito tiene por objeto dividir una red CAN en tantos segmentos como sea necesario, funcionando a diversas velocidades; es decir será necesario que se pueda ajustar la velocidad de comunicación y las distancias de cableado entre módulos en cada segmento, sin que esto tenga afectación sobre el rendimiento de otros segmentos. Esta característica se tiene que conseguir, sin soporte especial por parte del protocolo EuplaCAN, y de forma adicional la existencia de un módulo Gateway debe quedar transparente para el resto de los módulos de la red. Desde el punto de vista de módulo solo existiría un solo segmento.

Cada módulo Gateway consta de dos interfaces CAN, estando cada uno de ellos conectado a un segmento diferente. Este módulo deberá de mantener una lista de los nodos disponibles en cada uno de los interfaces, escuchando todo el tráfico producido en cada segmento de la red. Si se recibe una trama dirigida para un modulo que esta en el otro extremo de la red, éste copiará la trama tal cual en el otro segmento.

La dificultad de este diseño, radica en obtener y mantener las lista de nodos para cada uno de los interfaces, ya que el gateway tiene que darse cuenta si un nodo cambia de ubicación, si un nodo desaparece o si un nodo deja de existir.

La forma en que se ha resuelto el mantenimiento de las listas de nodos en los interfaces depende de que haya al menos un Host que realice encuestas periódicas de direcciones de nodo (tramas de ping) por la red. Este comportamiento introduce dos limitaciones; la primera es la existencia de al menos un modulo Host en la red y la segunda, reside en que éste tenga habilitada la capacidad de Hot PnP. Estas restricciones no han supuesto una alteración importante, debido a que redes que requieran nodos Gateway, son redes complejas, en las que la existencia de al menos un nodo Host, que supervise el buen funcionamiento de la red, es prácticamente obligatorio.

Debido a que el Gateway utiliza las tramas ping para mantener sus listas de nodos, deberán de ser copiadas, entre ambos interfaces indistintamente, actualizándose si es necesario cada lista de nodos dinámicamente.

3.6 Protocolo de comunicación EUPLACAN

3.6.1 Propósito del protocolo

La especificación EuplaCAN contiene los requerimientos de implementación mínimos para una red de comunicación de datos serie, para interconectar un conjunto de dispositivos electrónicos de adquisición de datos y control. Los dispositivos diseñados sobre esta especificación tendrán que tener la capacidad de convivir con otros dispositivos compatibles, en donde todos transmitirían sobre un único canal de señalización.

Este protocolo proveerá entre otras, las siguientes funcionalidades

- Identificación de módulos.
- Hot Plug and Play.
- Fácil detección y tratamiento de errores.
- Configuración remota de módulos
- Consulta de configuración de módulos
- Transmisión de datos por la red.

- Rutado de otros protocolos
- Persistencia.
- Sincronización.

3.6.1 Arquitectura de red y Tecnología CAN

La tecnología escogida para dar soporte al sistema ha sido CAN (Controller Area Network). Entre otras características CAN posee una transferencia de datos que va desde 5kbits/s hasta 1 Mb/s, transferencia de datos libre de error en configuraciones que soportan distancias de hasta 1,5 Km., fácil mantenimiento, bajo coste y sencillez de montaje del bus.

La especificación CAN esta disponible, ya programada, en diversos circuitos integrados, de manera que el usuario solo necesita conocer lo concerniente a los pequeños detalles de la tecnología de la comunicación con este bus. Los chips CAN son integrados sencillamente como periféricos inteligentes.

El hecho de que este bus esté virtualmente libre de problemas de mantenimiento y de uso, así como la caída rápida de los precios que los chips controladores han experimentado, ha hecho muy atractiva el uso de esta especificación para el sistema distribuido de adquisición de datos BRAVO 2000.

CAN es una especificación para comunicaciones serie asíncronas, la cual será usada para la comunicación de los módulos. Las capas más importantes de un protocolo de red (según ISO) son por una lado la capa física, que comprende la topología de la red y el enlace del bus; y por otro la capa de datos, la cual establece las bases de cómo se accede al medio de transmisión de datos, como se construye un mensaje y como está estructurado el protocolo de transmisión de datos.

La tecnología CAN, o mejor dicho, los chips que integran CAN, tienen implementado la capa física, la capa de acceso al medio(MAC) y la capa de enlace lógico(LLC). Las capas del 3 al 6 están vacías y en la capa 7(interfaces de aplicación) hay varias propuestas como SDS, CanOpen, DeviceNet, etc. Desde esta perspectiva EuplaCAN es una propuesta de interfaces de aplicación para sistemas distribuidos de adquisición de datos y control.

Las capas MAC y LLC utilizan un formato de trama que se emitirá por el bus y a partir de esta trama se establecerá un formato propietario con diversas funcionalidades. En pocas palabras, CAN proporciona la posibilidad de trasmitir de forma segura, paquetes con una cabecera de 29 bits y de una zona de datos de hasta 8 bytes.

EuplaCAN delega en capa CAN las funciones de:

- La generación de cadena de bits serie que tiene que ser transmitidas por la red.

- Ganar acceso a la red cuando un modulo necesita transmitir datos. Esto se consigue viendo cuando la red no esta ocupada. La forma en que CAN arbitra el bus permite un tratamiento de colisiones sin perdida de rendimiento.
- Chequeo de errores y retransmisión automática de paquetes incorrectos. CAN también puede atómicamente determinar cuando un nodo tiene errores repetitivos llevando el nodo fuera de línea para proteger al resto de la red.

3.6.2 Especificación del protocolo EUPLACAN

Las tareas de las que EuplaCAN se encarga son las siguientes:

- Definir el modo en que se usaran los 29 bits de cabecera y los 8 bytes de datos de las tramas CAN.
- Proveer un método para enviar paquetes específicos de cada modulo y fabricante.
- Definir cómo interpretarse la información transmitida en una trama CAN simple de 8 bytes
- Definir métodos para transmitir información que no puede enviarse en una trama CAN simple de 8 bytes.
- Definir un conjunto común de mensajes para solicitar datos, enviar comandos a módulos y el reconocimiento de comandos, datos y peticiones.
- Definir métodos para la implementación de otros protocolos sobre EuplaCAN.

En definitiva, EuplaCAN define la forma en que los nodos interactúan entre sí en una red CAN.

En EuplaCAN, las comunicaciones son punto a punto y por tanto, cada nodo debe poseer una dirección única que le identifique ante el bus. Cada mensaje transmitido en el bus es para un nodo concreto.

En ese intercambio de mensajes entre nodos, se observan dos tipos de comportamientos diferentes. Un comportamiento síncrono y otro asíncrono. Al enviarse un mensaje de forma síncrona, se espera recibir una respuesta, que de no satisfacerse en un periodo finito de tiempo, generaría un error en aquél modulo que envió la solicitud. En el comportamiento asíncrono un mensaje se envía a otro nodo sin esperar una notificación.

La comunicación síncrona, permite modelar la solicitud de información o la ejecución de comandos sobre un módulo, en el que el primer mensaje sería la solicitud y en la contestación vendría empaquetada la información solicitada o simplemente un reconocimiento de que el comando ha sido ejecutado con éxito. La naturaleza de la información o comando que se solicita se modela con el concepto de “Servicio”.

La comunicación asíncrona permite modelar una transmisión periódica de datos de una manera más eficiente (ya que no se genera un paquete de respuesta por cada paquete de datos emitido), así como una notificación de errores o cambio de estado

más natural (ya que éstos son de carácter asíncrono). La naturaleza de la información transmitida también se modela teniendo en cuenta el concepto de servicio.

Con estos dos puntos de vista, se obtiene tres tipos distintos de mensajes o tramas, que caracterizan cada tipo de intercambio de información:

- **Tramas de Petición (Request):** Su funcionalidad será la de realizar solicitudes.
- **Tramas de Reconocimiento(ACK o NAK):** Serán las tramas de respuesta a una trama previa de tipo Request. Las tramas NAK se generan cuando, o bien no se conoce el servicio solicitado o la trama Request ha generado algún tipo de error. Tanto las tramas ACK como las tramas NAK, son respuestas a una solicitud de petición de servicio (una es una respuesta de aceptación y la otra de negación respectivamente), y por tanto, en el campo servicio se debe indicar el servicio indicado en la trama Request.
- **Tramas de Transmisión (Transmit):** Simplemente transportan información sin espera de una trama de reconocimiento.

Ahora bien, cada uno de estos tres tipo admite dos variantes que especifican el carácter de la trama. De esta forma se pueden distinguir tramas de carácter Genérico y tramas de carácter Especifico. Una trama es de carácter Genérico o de carácter Especifico, en función de sí forma parte de la especificación general de EuplaCAN o si pertenece a la especificación particular de un módulo. Las tramas de carácter genérico, deben ser implementadas por todos los módulos EuplaCAN, mientras que las tramas especificas no.

En función del servicio requerido, un mensaje puede transmitir más o menos información. Al resto de información que puede empaquetarse en mensaje EuplaCAN, se denominará campos de datos. La existencia y tipo de campos de datos en una trama son dependientes del tipo de trama, carácter de la misma y servicio.

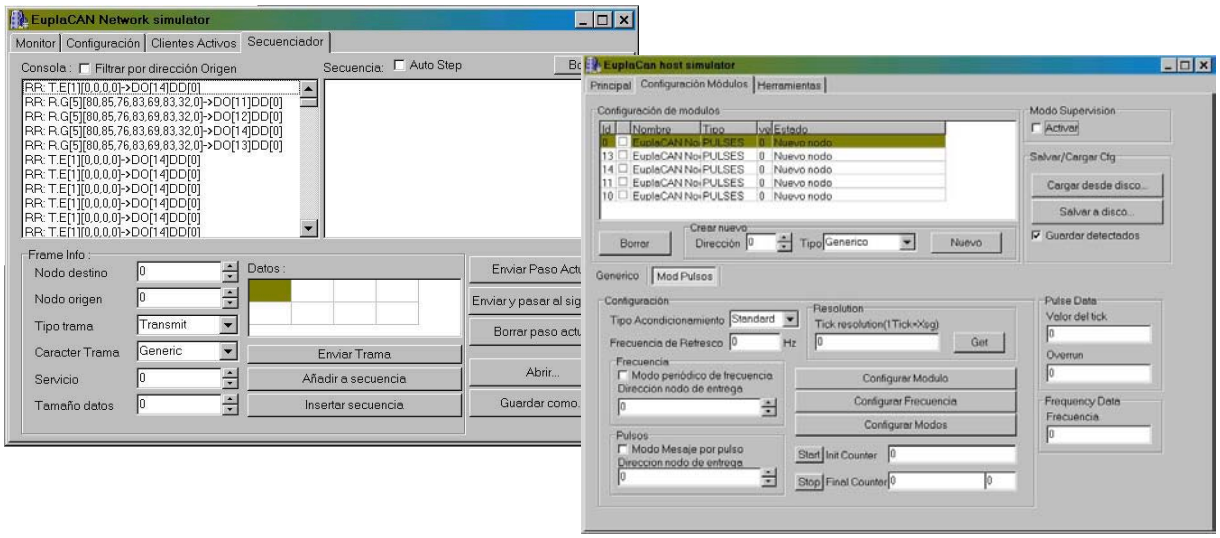
En conclusión EuplaCAN se reduce a la especificación de tres tipos de tramas(Request, Ack o NAK y Transmit), pudiendo ser cada una de ellas de carácter Genérico o de carácter Especifico siendo la lógica de la comunicación de Peticion-Respuesta o Transmisión asíncrona.

3.5 Software de control

Como anteriormente se ha comentado toda la red está controlada por un modulo principal que actúa como servidor de datos a la aplicación cliente, mediante tecnología TCP/IP, que se encargará de la monitorización y control del modulo principal de forma remota. Esta aplicación cliente actuará como interface de usuario y permitirá:

- Identificar los módulos instalados en la red.
- Configurar de forma especifica cada uno de los módulos.
- Autodetección de problemas en la red, así como su identificación y resolución.

- Gestión de la hipótesis de un ensayo, desde establecer los parámetros de la hipótesis hasta la generación de informes.
- Control y monitorización gráfica del ensayo.
- Control y monitorización gráfica en formato SCADA del proceso industrial instrumentado.
- Preprocesado de datos, desde el acondicionamiento típico de calibración y filtrado de la señal adquirida, hasta la generación de nueva variables calculadas a partir de los datos obtenidos en la adquisición.
- Postprocesado de las señales obtenidas. Análisis en el dominio del tiempo y de la frecuencia. Así como cálculos estadísticos entre las señales obtenidas, como correlaciones, regresiones, variaciones, etc..
- Gestión del almacenamiento de información para su posterior consulta.
- Control de forma remota a través de LAN o incluso a través de Internet.



Referencias

1. Beck M., Bhome H,.. (1998) Linux Kernel Book Wiley
2. Bosch R. (1991) CAN Specification 2.0 Part B. Robert Bosch GmbH
3. Card R.,Dumas E., Mével F. (1997)Linux Kernel Internals Addison Wesley
4. Enslow P.H. (1978). What is a distributed system. Computer, January 1978, pp. 13–21.
5. Fox M.S. (1981). An organizational view of distributed systems. IEEE Trans. Vol.-SMC–11, No. 1
6. Phillips Semiconductor (1994) SJA Stand Alone CAN Controller
7. Stallings W. (1995). Data and computer communications. Prentice–Hall. ISBN 0–13–326828–4.

Understanding Perfect Failure Detectors *

Mikel Larrea

Universidad del País Vasco
20018 San Sebastián, Spain
mikel.larrea@si.ehu.es

Abstract. Chandra and Toueg [1], Fromentin et al. [3] and Sabel and Marzullo [7], respectively, stated that the weakest failure detector for any of Non-Blocking Atomic Commitment, Terminating Reliable Broadcast and Leader Election is the *Perfect* failure detector $\mathcal{P}$. More recently, Guerraoui [5] presented a counterexample of those results, exhibiting a failure detector called *Marabout* ($\mathcal{M}$) that is incomparable to $\mathcal{P}$ and yet solves those problems. $\mathcal{M}$ is a failure detector that predicts the future, and cannot be implemented even in a completely synchronous system, being useless from a practical point of view.
In this paper we present three new perfect failure detector classes as useful alternatives to $\mathcal{P}$ and $\mathcal{M}$. All our classes are weaker than $\mathcal{P}$. Furthermore, two of them are also weaker than $\mathcal{M}$, and yet solve Non-Blocking Atomic Commitment, Terminating Reliable Broadcast and Leader Election. Interesting enough, our failure detector classes are implementable whenever $\mathcal{P}$ is implementable (e.g., in a synchronous system), which is not the case with $\mathcal{M}$.

1 Introduction

Non-Blocking Atomic Commitment, *Terminating Reliable Broadcast* and *Leader Election* are three fundamental agreement problems in reliable distributed computing. This paper discusses the use of perfect failure detectors for solving these problems in distributed systems where channels are reliable and processes can fail by crashing.

In Non-Blocking Atomic Commitment, the processes must agree on the outcome of distributed transactions: commit or abort. The outcome depends on the votes of the processes: yes or no. The decision must be commit if all processes vote yes and no process crashes [8]. In Terminating Reliable Broadcast, the processes need to agree on whether to deliver a message broadcast by some specific sender process, or to deliver a default message. The processes must deliver the message broadcast by the specific sender if the sender does not crash [6]. In Leader Election, the processes

* Research partially supported by the Spanish Research Council (CICYT), under grants TIC99-0280-C02-02 and TIC2001-1586-C03-01.

must elect a leader and make sure to avoid any disagreement about which process is leader at any given time. If the current leader crashes, a new leader must be elected [7].

Chandra and Toueg [1], Fromentin et al. [3] and Sabel and Marzullo [7], respectively, stated that the weakest failure detector for any of Non-Blocking Atomic Commitment, Terminating Reliable Broadcast and Leader Election is the *Perfect* failure detector $\mathcal{P}$. Failure detector $\mathcal{P}$ ensures that (a) eventually, every correct process permanently suspects every crashed process, and (b) no process is suspected before it crashes. Hence, according to [1, 3, 7], to solve any of those agreement problems, the knowledge about failures provided by $\mathcal{P}$ is *sufficient* and *necessary*.

More recently, Guerraoui [5] presented a counterexample of those results, exhibiting a failure detector called *Marabout* (denoted $\mathcal{M}$) that is incomparable to $\mathcal{P}$ and yet solves those problems. Failure detector $\mathcal{M}$ ensures that (a) every incorrect process is permanently suspected by every correct process, and (b) no process is suspected unless it crashes. Clearly, $\mathcal{M}$ is a failure detector that predicts the future, and cannot be implemented even in a completely synchronous system, being useless from a practical point of view.

1.1 Our Results

In this paper we present three new perfect failure detector classes as useful alternatives to $\mathcal{P}$ and $\mathcal{M}$. All our classes are weaker than $\mathcal{P}$. Furthermore, two of them are also weaker than $\mathcal{M}$, and yet solve Non-Blocking Atomic Commitment, Terminating Reliable Broadcast and Leader Election. Thus, like Guerraoui we show that $\mathcal{P}$ is not the weakest failure detector to solve any of those problems. Interesting enough, our failure detector classes are implementable whenever $\mathcal{P}$ is implementable (e.g., in a synchronous system), which is not the case with $\mathcal{M}$.

The rest of the paper is organized as follows. In Section 2 we present the system model we will consider in the paper. In Section 3 we present the new classes of perfect failure detectors, studying the relationship between them and $\mathcal{P}$ and $\mathcal{M}$. In Section 4 we show first how to solve Consensus, and then Non-Blocking Atomic Commitment, Terminating Reliable Broadcast and Leader Election using our perfect failure detectors. In Section 5, we give some general remarks. Finally, Section 6 concludes the paper.

2 System Model

We consider an asynchronous distributed system augmented with the failure detector abstraction [1]. The system consists of a finite set of n processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$, that communicate only by sending and receiving messages. Every pair of processes is connected by a reliable communication channel.

To simplify the presentation of the model, we assume the existence of a discrete global clock. This is a merely fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

2.1 Failures and Failure Patterns

Processes execute deterministic algorithms and can fail by *crashing*, that is, by prematurely halting. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not recover, that is, $\forall t : F(t) \subseteq F(t+1)$. We define $crashed(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $correct(F) = \Pi - crashed(F)$. If $p \in crashed(F)$, we say p *crashes in* F and if $p \in correct(F)$, we say p *is correct in* F. We consider only failure patterns F such that at least one process is correct, that is, $correct(F) \neq \emptyset$.

2.2 Failure Detectors

Each failure detector module outputs the set of processes that it currently suspects to have crashed. A *failure detector history* H is a function from $\Pi \times \mathcal{T}$ to 2^{Π}. $H(p,t)$ is the value of the failure detector module of process p at time t. If $q \in H(p,t)$, we say that p *suspects* q *at time* t *in* H. Note that the failure detector modules of two different processes need not agree on the set of processes that are suspected to have crashed, that is, if $p \neq q$, then $H(p,t) \neq H(q,t)$ is possible.

Informally, a failure detector $\mathcal{D}$ provides (possibly incorrect) information about the failure pattern F that occurs in an execution. Formally, *failure detector* $\mathcal{D}$ is a function that maps each failure pattern F to a set of failure detector histories $\mathcal{D}(F)$. This is the set of all failure detector histories that could occur in executions with failure pattern F and failure detector $\mathcal{D}$.

3 A Family of Perfect Failure Detector Classes

We now state a completeness property and four accuracy properties that a failure detector $\mathcal{D}$ may satisfy. Consider first the following completeness property:

- *Strong Completeness.* Eventually every process that crashes is permanently suspected by every correct process. Formally, $\mathcal{D}$ satisfies Strong Completeness if:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall p \in crashed(F), \forall q \in correct(F), \forall t' \geq t : p \in H(q, t').$$

Consider now the following four accuracy properties:

- *Strong Accuracy 1.* No process is suspected before it crashes. Formally, $\mathcal{D}$ satisfies Strong Accuracy 1 if:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p, q \in \Pi - F(t) : p \notin H(q, t).$$

- *Strong Accuracy 2.* No process is suspected *by any correct process* before it crashes. Formally, $\mathcal{D}$ satisfies Strong Accuracy 2 if:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p \in \Pi - F(t), \forall q \in correct(F) : p \notin H(q, t).$$

- *Strong Accuracy 3.* No *correct* process is ever suspected. Formally, $\mathcal{D}$ satisfies Strong Accuracy 3 if:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p \in correct(F), \forall q \in \Pi : p \notin H(q, t).$$

- *Strong Accuracy 4.* No *correct* process is ever suspected *by any correct process.* Formally, $\mathcal{D}$ satisfies Strong Accuracy 4 if:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p, q \in correct(F) : p \notin H(q, t).$$

Combining Strong Completeness with each one of the previous accuracy properties, we obtain four perfect failure detector classes:

- The class P_1, satisfying Strong Completeness and Strong Accuracy 1. This class corresponds to the class of *Perfect* failure detectors $\mathcal{P}$ defined by Chandra and Toueg in [1].
- The class P_2, satisfying Strong Completeness and Strong Accuracy 2.
- The class P_3, satisfying Strong Completeness and Strong Accuracy 3.
- The class P_4, satisfying Strong Completeness and Strong Accuracy 4.

Clearly, P_1 is the strongest class among the previous four classes, since it does not allow any incorrect suspicion. On the other hand, the other three classes do allow some incorrect suspicion. Let p and q be two processes. Figure 1 presents the possible mistakes made by p regarding q for each one of the four classes. For any incorrect process, it is assumed that it has not crashed yet. We also consider as a mistake the premature suspicion of an incorrect process that has not crashed yet, even if it is known that it will eventually crash.

	p **incorrectly suspects** q			
FD class	p and q correct	p correct, q incorrect	p incorrect, q correct	p and q incorrect
P_1	No	No	No	No
P_2	No	No	**Yes**	**Yes**
P_3	No	**Yes**	No	**Yes**
P_4	No	**Yes**	**Yes**	**Yes**

Fig. 1. Mistakes that can be made by each failure detector class.

The class P_1 does not allow any kind of mistake. On the other hand, the class P_2 allows incorrect processes to incorrectly suspect other processes (p incorrect). Similarly, the class P_3 allows incorrect processes to be prematurely suspected (q incorrect). Finally, the class P_4 allows the two kinds of mistakes of classes P_2 and P_3 (p or q incorrect).

3.1 Relation between P_1, P_2, P_3, and P_4

We have the following relationships:

1) $P_1 \subset P_2$. By definition, every failure detector in P_1 is also in P_2. In other words, P_1 can be transformed into P_2 (actually, any implementation of P_1 also implements P_2). On the other hand, P_2 cannot be transformed into P_1: with P_2, an incorrect process p can suspect a correct process q, violating the accuracy property of P_1. Thus, P_2 is strictly weaker than P_1.
2) $P_1 \subset P_3$. By definition, every failure detector in P_1 is also in P_3. In other words, P_1 can be transformed into P_3 (actually, any implementation of P_1 also implements P_3). On the other hand, P_3 cannot be transformed into P_1: with P_3, a correct process p can prematurely suspect an incorrect process q before q has crashed, violating the accuracy property of P_1. Thus, P_3 is strictly weaker than P_1.

3) P_2 and P_3 are incomparable. Not every failure detector in P_2 is in P_3: for example, with P_2 an incorrect process p can suspect a correct process q, which violates the accuracy property of P_3. Similarly, not every failure detector in P_3 is in P_2: for example, with P_3 a correct process p can prematurely suspect an incorrect process q before q has crashed, which violates the accuracy property of P_2.
4) $P_2 \subset P_4$. By definition, every failure detector in P_2 is also in P_4. In other words, P_2 can be transformed into P_4 (actually, any implementation of P_2 also implements P_4). On the other hand, P_4 cannot be transformed into P_2: with P_4, a correct process p can prematurely suspect an incorrect process q before q has crashed, violating the accuracy property of P_2. Thus, P_4 is strictly weaker than P_2.
5) $P_3 \subset P_4$. By definition, every failure detector in P_3 is also in P_4. In other words, P_3 can be transformed into P_4 (actually, any implementation of P_3 also implements P_4). On the other hand, P_4 cannot be transformed into P_3: with P_4, an incorrect process p can suspect a correct process q, violating the accuracy property of P_3. Thus, P_4 is strictly weaker than P_3.

3.2 Marabout is a Subclass of P_3 (and P_4)

In [5], Guerraoui defined a failure detector called *Marabout* and denoted $\mathcal{M}$. Informally, $\mathcal{M}$ is a failure detector that *predicts* the crashes of processes in an accurate manner, but does not say when the crashes will actually occur. Formally, the class $\mathcal{M}$ of failure detectors satisfies the following completeness and accuracy properties:

- *Perpetual Completeness.* Every incorrect process is permanently suspected by every correct process. More precisely:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p \in crashed(F), \forall q \in correct(F) : p \in H(q, t').^{1}$$

- *Perpetual Accuracy.* No process is suspected unless it crashes. More precisely:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p, q \in correct(F) : p \notin H(q, t).^{2}$$

[1] This definition corresponds to the *Psychic* Strong Completeness property defined in [2].
[2] This definition corresponds to the *Psychic* Strong Accuracy property defined in [2]. It also corresponds to the Strong Accuracy 4 property defined in this section.

It is shown in [5] that classes $\mathcal{M}$ and P_1 – Chandra-Toueg's $\mathcal{P}$ – are incomparable. Intuitively, P_1 provides perfect failure *detection*: it outputs accurate information about past crashes. In contrast, $\mathcal{M}$ provides perfect failure *prediction*: it outputs accurate information about future crashes. These are incomparable kinds of knowledge about failures. By a similar reasoning, $\mathcal{M}$ and P_2 are also incomparable.

However, $\mathcal{M}$ can be compared to P_3 and P_4. Note that, by definition, every failure detector in $\mathcal{M}$ is also in P_3. Hence, the class $\mathcal{M}$ is a subclass of P_3, i.e., $\mathcal{M} \subset P_3$. Therefore, we also have that $\mathcal{M} \subset P_4$. In other words, $\mathcal{M}$ can be transformed into P_3 and P_4 (actually, any implementation of $\mathcal{M}$ also implements P_3 and P_4). On the other hand, neither P_3 nor P_4 can be transformed into $\mathcal{M}$: with P_3 and P_4, a correct process p may not suspect an incorrect process q from the beginning but after some time, maybe after q has crashed, violating the completeness property of $\mathcal{M}$. Thus, P_3 and P_4 are strictly weaker than $\mathcal{M}$.

3.3 Implementability of Perfect Failure Detectors

In a synchronous system (in which there are known upper bounds on the transmission delay of messages and the relative speeds of processes), a failure detector of class P_1 can be implemented. In such a system, one can build a simple timeout-based algorithm that reliably detects the failure of processes. Hence, it is straightforward to see that classes P_2, P_3, and P_4 can also be implemented in a synchronous system, since they are weaker than P_1. In contrast, $\mathcal{M}$ cannot be implemented even in a synchronous system, since no algorithm can predict the future.

4 Solving Agreement Problems Using Perfect Failure Detectors

In this section, we show how to solve Consensus, Non-Blocking Atomic Commitment, Terminating Reliable Broadcast, and Leader Election using a perfect failure detector as weak as possible from those presented in the previous section. We already know that all these problems can be solved using P_1, i.e., Chandra-Toueg's $\mathcal{P}$ [1, 3, 7]. However, we show here that P_3 suffices for solving Terminating Reliable Broadcast, while for the other three problems P_4 – the weakest class presented in the previous section – is sufficient.

4.1 Solving Consensus Using P_4

In the Consensus problem, each process proposes initially a value, and all correct processes must reach an irrevocable decision on some common value that is equal to one of the proposed values. Formally, the Consensus problem is defined in terms of two primitives, *propose* and *decide*. When a process executes *propose*(v), we say that it *proposes* v; similarly, when a process executes *decide*(v), we say that it *decides* v. The *Consensus* problem is specified as follows:

- **Agreement.** No two correct processes decide differently.
- **Termination.** Every correct process eventually decides some value.
- **Validity.** If a process decides v, then v was proposed by some process.
- **Uniform integrity.** Every process decides at most once.

The agreement property allows faulty processes to decide differently from correct processes. This fact can be sometimes undesirable as it does not prevent an incorrect process to propagate a different decision throughout the system before crashing. In the *Uniform Consensus* problem, agreement is defined by the following property, which enforces the same decision on any process that decides:

- **Uniform agreement.** No two processes (correct or faulty) decide differently.

It has been shown in [4] that any algorithm that solves Consensus using a failure detector of class $\diamond\mathcal{S}$, also solves Uniform Consensus.

Lemma 1. *P_4 can be used to solve Uniform Consensus.*

Proof. (Sketch) Chandra-Toueg's $\diamond\mathcal{S}$-Consensus protocol [1] is correct with a failure detector of class P_4. To see it, note that P_4 is a subclass of $\diamond\mathcal{S}$, since both satisfy Strong Completeness, and Strong Accuracy 4 (satisfied by P_4) involves Eventual Weak Accuracy (satisfied by $\diamond\mathcal{S}$).

4.2 Solving Non-Blocking Atomic Commitment Using P_4

In the Atomic Commitment problem, processes have to *decide* on an outcome value *commit* or *abort*. The outcome value depends on *votes* (*yes* or *no*) *proposed* by the processes. Every process proposes exactly one value. We consider the *non-blocking* version of the problem, in which every correct process eventually decides even if some process has crashed [8]. The *Non-Blocking Atomic Commitment* problem is specified as follows:

- **Agreement.** No two processes decide differently.
- **Termination.** Every correct process eventually decides.
- **Abort-validity.** *Abort* is the only possible decision if some process votes *no*.
- **Commit-validity.** *Commit* is the only possible decision if every process votes *yes* and no process crashes.

Lemma 2. *P_4 can be used to solve Non-Blocking Atomic Commitment.*

Proof. (Sketch) We give here a brief description of a protocol that solves Non-Blocking Atomic Commitment using a failure detector of class P_4. The protocol works as follows. Initially, every process p sends its vote (*yes* or *no*) to all processes (including itself). Then, every process p waits for the vote or the suspicion of all processes. If some process is suspected or some vote is *no*, then p sets its estimate to *abort*; else (all votes are *yes* and no process is suspected) p sets its estimate to *commit*. Finally, every process p executes a Consensus with its estimate as its first proposition. The decision of the Consensus will be the outcome of the Non-Blocking Atomic Commitment problem.

This protocol satisfies all the properties of Non-Blocking Atomic Commitment. Agreement and termination follow from agreement and termination of Consensus respectively. Abort-validity follows from the fact that if a process votes *no*, then by the protocol no process will propose *commit*, and thus the only possible decision of Consensus will be *abort*. Finally, commit-validity follows from the fact that if every process votes *yes* and no process crashes, then by the protocol all processes will propose *commit*, and thus the only possible decision of Consensus will be *commit*.

4.3 Solving Terminating Reliable Broadcast Using P_3

In the Terminating Reliable Broadcast problem, a distinguished *sender* process, noted p, is supposed to broadcast a message m from a set M of possible messages: we note $sender(m) = p$. All processes are supposed to deliver, either that message m, or a message $F_p \notin M$ (F_p states that the sender p is faulty) [6]. Terminating Reliable Broadcast is similar to *Reliable Broadcast*, except that it requires that every correct process always delivers a message, even if the sender crashes before broadcasting a message. Given a sender process p and a message m, TRB_p is specified as follows:

- **Agreement.** No two correct processes deliver two different messages.

- **Termination.** Every correct process eventually delivers exactly one message.
- **Validity.** If p is correct and p broadcasts a message m, then p delivers m.
- **Integrity.** If a correct process delivers a message m then $sender(m) = p$, and if $m \neq F_p$, then m was previously broadcast by p.

We call *Terminating Reliable Broadcast* here the problem that gathers multiple instances of TRB_p: one for every process $p \in \Pi$.

Lemma 3. *P_3 can be used to solve Terminating Reliable Broadcast.*

Proof. (Sketch) We give here a brief description of a protocol that solves Terminating Reliable Broadcast using a failure detector of class P_3. We assume that, following TRB_p, every correct process p sends msg_p to all processes. The protocol works as follows. Initially, every process p waits for the message or the suspicion of all processes. If p receives a message msg_q from process q then p sets $estimate_{p,q}$ to $< yes, msg_q >$; else (p suspects q) p sets $estimate_{p,q}$ to $< no, F_q >$. Finally, every process p executes $Consensus_q$ with $estimate_{p,q}$ as its first proposition. If the decision of $Consensus_q$ is $< yes, msg_q >$ then p delivers msg_q; else p delivers F_q. Hence, n independent Consensus are executed, one for each process. Note also that every process manages n estimates, one for each process.

This protocol satisfies all the properties of Terminating Reliable Broadcast. Agreement and termination follow from agreement and termination of Consensus respectively. Integrity follows directly from the protocol. Finally, validity follows from the fact that if a process p is correct, then no process suspects it (by the accuracy property of P_3). Hence, all correct processes will start $Consensus_p$ with $< yes, msg_p >$ as estimate, and thus the only possible decision of $Consensus_p$ is $< yes, msg_p >$. By the protocol, every correct process will deliver msg_p. Note that if some process p suspects another process q, this means – by P_3 – that q is incorrect (i.e., q will eventually crash). Thus, we do not violate validity if msg_q is not delivered.

Remark. Note that this protocol does not work with a failure detector of class P_4: with P_4, if an incorrect process p suspects a correct process q, validity can be violated if the decision of $Consensus_q$ is $< no, F_q >$, since q is correct but F_q is delivered instead of the message msg_q. The same scenario can happen with a failure detector of class P_2.

4.4 Solving Leader Election Using P_4

In the Leader Election problem [7], at any time at most one process considers itself the *leader*, and if a leader crashes, a new leader must eventually be elected. To precisely capture the notion of leadership, we assume that every process has a local copy of a distributed variable denoted by *leader*. The copy of *leader* at a process p is denoted by $leader_p$, and for any process p, $leader_p \in \{true, false\}$. We say that a process p is leader at a time t if p has not crashed by time t and $leader_p = true$. The *Leader Election* problem is specified as follows:

- **Uniform agreement.** No two processes can be leader at the same time.
- **Termination.** At any time, if there is no leader, a leader is eventually elected.

Lemma 4. *P_4 can be used to solve Leader Election.*

Proof. (Sketch) We give here a brief description of a protocol that solves Leader Election using a failure detector of class P_4. The protocol works as follows. Initially, every process p sets $leader_p$ to *false*. Every process p queries its failure detector module and sets its estimate to the lowest index among the indexes of the processes not suspected. Finally, every process p executes a Consensus (we require *Uniform* Consensus) with its estimate as its first proposition. The process with index equal to the decision of the Consensus sets *leader* to *true* (i.e., if the decision is k, then process k sets $leader_k$ to *true*).

This protocol satisfies all the properties of Leader Election. Uniform agreement and termination follow from uniform agreement and termination of Consensus respectively.

5 Remarks

5.1 On the Use of Marabouts

With respect to the Marabout failure detector $\mathcal{M}$, it is easy to see that with $\mathcal{M}$ the sets of suspected processes of all correct processes are always identical, which is by itself a consensus. Thus, not surprisingly the protocols proposed in [5] to solve Non-Blocking Atomic Commitment, Terminating Reliable Broadcast, and Leader Election using $\mathcal{M}$ do not rely on a Consensus protocol but solely on $\mathcal{M}$, because $\mathcal{M}$ itself provides the consensus. In contrast, our protocols using P_3 or P_4 rely on a Consensus

protocol, since correct processes do not know *when* their sets of suspected processes are identical, even if it is known that they will eventually (and permanently) be identical.

5.2 Solving Consensus in a Bounded Number of Rounds

Failure detector classes P_2 and P_4 allow an incorrect process p to suspect a correct process q before crashing. Since we do not know when p will crash, these classes do not provide more useful information about failures than the *Eventually Perfect* class $\diamond\mathcal{P}$ [1].

Consider now the following two accuracy properties that a failure detector $\mathcal{D}$ may satisfy:

- *Weak Accuracy 1.* Some correct process is never suspected. Formally, $\mathcal{D}$ satisfies Weak Accuracy 1 if:

  $$\forall F, \forall H \in \mathcal{D}(F), \exists p \in \mathit{correct}(F), \forall t \in \mathcal{T}, \forall q \in \Pi - F(t) : p \notin H(q,t).$$

- *Weak Accuracy 2.* Some correct process is never suspected by any correct process. Formally, $\mathcal{D}$ satisfies Weak Accuracy 2 if:

  $$\forall F, \forall H \in \mathcal{D}(F), \exists p \in \mathit{correct}(F), \forall t \in \mathcal{T}, \forall q \in \mathit{correct}(F) : p \notin H(q,t).$$

Combining Strong Completeness with each one of these two accuracy properties, we obtain two classes of *strong* failure detectors:

- The class S_1, satisfying Strong Completeness and Weak Accuracy 1. This class corresponds to the class of *Strong* failure detectors $\mathcal{S}$ defined by Chandra and Toueg in [1].
- The class S_2, satisfying Strong Completeness and Weak Accuracy 2.

Clearly, S_2 is weaker than S_1. S_1 does not allow any incorrect suspicion with respect to some correct process p, while S_2 allows incorrect processes to incorrectly suspect p. In [1], Chandra and Toueg propose a protocol that solves Consensus using S_1 in a bounded number of rounds (n rounds, with n the number of processes). Note that this protocol does not work with S_2, neither with P_2 or P_4. The problem comes from the behavior of incorrect processes, more precisely from the incorrect suspicions they can make. With S_2, P_2, and P_4 the number of rounds of the Consensus protocol can only be bounded after the crash of all incorrect processes, and not from the beginning of the protocol. Thus, we have to use protocols with an a priori unbounded number of rounds. For

example, Chandra-Toueg's $\diamond\mathcal{S}$-Consensus protocol, which uses the rotating coordinator paradigm, works with any of the failure detector classes $P_1, P_2, P_3, P_4, S_1, S_2$. In other words, P_2 and P_4 are not subclasses of S_1, while P_1 and P_3 are. Obviously, $P_1, P_2, P_3, P_4, S_1, S_2$ are all subclasses of $\diamond\mathcal{S}$.

6 Conclusions

In this paper we have presented three new perfect failure detector classes. Our classes are useful alternatives to the classes $\mathcal{P}$ and $\mathcal{M}$, proposed by Chandra and Toueg [1] and Guerraoui [5] respectively. More precisely, two of them are weaker than $\mathcal{P}$ and $\mathcal{M}$, and yet can be used to solve Non-Blocking Atomic Commitment, Terminating Reliable Broadcast and Leader Election. Thus, like Guerraoui we show that $\mathcal{P}$ is not the weakest failure detector to solve any of those problems, contradicting existing previous results [1, 3, 7]. Interesting enough, our failure detector classes are implementable whenever $\mathcal{P}$ is implementable, which is not the case with $\mathcal{M}$.

Acknowledgments

We are grateful to Antonio Fernández for his valuable comments on a draft of this paper.

References

1. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
2. A. Fernández. Extended taxonomy of unreliable failure detectors. Unpublished manuscript, Departamento de Arquitectura y Tecnología de Computadores, Universidad Politécnica de Madrid, June 1998.
3. E. Fromentin, M. Raynal, and F. Tronel. About classes of problems in asynchronous distributed systems with process crashes. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 470–477, Austin, Texas, May 1999.
4. R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG)*, pages 87–100. LNCS, Springer-Verlag, September 1995.
5. R. Guerraoui. On the hardness of failure-sensitive agreement problems. *Information Processing Letters*, 79:99–104, 2001.
6. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. Addison-Wesley, 2nd edition, 1993.

7. L. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, Computer Science Department, February 1995.
8. D. Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–142, Ann Arbor, Michigan, April 1981.

Tactics for the Iterative Elimination of Communications in Concurrent Programs

Francesc Babot, Miquel Bertran, August Climent, Miquel Nicolau, and Josep M. Muixí

Informàtica La Salle, Universitat Ramon Llull, Barcelona
{fbabot,miqbe,augc,miqueln,jmmuixi}@salleURL.edu

Abstract. A set of tactics for the formal elimination of multiple synchronous communication statements of imperative concurrent programs is presented. They guarantee that the transformation of the program always corresponds to the application, as reductions, of a sequence of laws which preserve program equivalence in a well defined sense. Although it can be extended to more general program forms, only a special subset of programs is treated. The programs are expressed in a subset of SPL, a good representative of imperative notations for concurrent, distributed, and reactive programs, explained in the books of Manna and Pnueli, and used within the Stanford Temporal Prover. One of the envisaged applications of the tactic is formal verification within the framework of a suitable environment, in conjunction with program verifiers and model checkers. Then a program with communications would be transformed into a communication-free program whose verification would be simpler. However, much work is still needed to develop such an environment.

1 Introduction

The long term aim of this work is the construction of a system for the simplification of programs with inner parallelism and synchronous communication statements. Simplification meaning deriving equivalent programs where inner synchronous communication would have been replaced by assignment statements within a different parallelism structure. In general these programs would be simpler and easier to verify.

The basic constituent steps of formal communication simplification are simple transformations, which have to correspond to congruences or other equivalence preserving relations between program statements. The soundness and meaning of them has to be mathematically established. Some sets of equivalence laws for concurrent programs are available in the literature, particularly in the area of process algebras. The books of Hoare and Milner [Hoa85,Mil89] contain some such sets. In the area of static analysis, transformations have been reported in [Bens00] and methods to derive parallel codes have been treated in [Chin00]. The works of Lamport and Hooman, reported in [Lam94] and [Hoo94] respectively, are related to the topic. In many cases some of the laws are neither simple nor intuitive. In addition, we remark that all these approaches are different to the

one reported in the present work. The general problem of program analysis for communication elimination and, at the syntactic level, of *imperative programs* is not treated in the above works and, up to our present knowledge, has not been dealt with in the literature either. The *simple programming language* SPL is introduced and used in the books of Manna and Pnueli [MaPn91,MaPn94], and some simple laws are justified there under the assumption that synchronous communication is not used. Otherwise, some of the intuitive laws do not hold. This problem is by no means restricted to SPL, it is shared by other notations for concurrent and distributed systems.

We base the present work on a set of simple relations which have been mathematically justified in [Ber01] for a restricted version of SPL. For self-containment reasons, this set of laws is given here without proof, since all the tactics are based on them. It is important to remark that for the applicability of the proper communication elimination laws this set of simple relations is a requirement. This point is clarified in the above references, and it is shown that one needs then to work on a restricted version of SPL. The restriction has no importance in practice. It amounts to prefix a skip statement to any synchronous communication statement.

Some initial work on the tactics was reported in [Bab01]. We concentrate here in the tactic which eliminates matching synchronous communication statements from a program whose main structure corresponds to a parallel statement. Only a subset of programs is contemplated, the programs whose synchronous communications never appear within selection statements. The concept of tactic has been taken from the seminal work of LCF [GrMlWa79]. Our tactic has an iterative structure. Its step consists of a subtactic that eliminates a single pair of matching synchronous communication statements. The only transformations performed on the original program are done by the application of laws belonging to the justified set, after successfull unification. However, in order to transform the program into a form where unification would succeed, some preliminary subtactics identify key substatement and perform the required transformations by the application of simple laws.

After a section devoted to the required background: concepts, notation and laws, the presentation proceeds to explain the tactic which eliminates a pair of communication statements. Then the main iterative tactic is covered in another section which also contains the required definitions. The communication ends with a section on conclusions and further work.

2 General Background

Restricted SPL is the notation assumed throughout this work. It satisfies the laws given in this section without justification. As in SPL, the semantics is defined in terms of finite transition systems FTSs. The reader is referred to [MaPn91,MaPn94] and [Ber01] for a detailed coverage of the semantics of SPL and of its restricted version. The laws correspond to congruence and refinement

relations between statements, and hold for a restricted version of SPL. We review some concepts first.

A *program context*, represented as $P[S]$ with some abuse of notation, is a program P one of whose statements S is marked or highlighted. This concept is needed for the definition of congruence. In some design scenarios where synchronous communications are involved we need the following more flexible notion.

Definition 1 (Flexible program context) *Let S be a statement having synchronous communication operations $com(\alpha_1), \ldots, com(\alpha_n)$ with channels α_1, $\ldots$, α_n which are not local to S. These communication operations have to match synchronous communication operations in statements parallel to S in any program context $F[S]$. We say that a program context $F[S]$ is a* flexible *program context with respect to the non-local communication operations of S when the statements parallel to S in $F[S]$ are disjoint with those in S and offer communication operations matching the external communication operations in S in such a way that no specific order is imposed upon them.*

The following is an example:

$$F[S] \;=\; S \;||\; \textbf{while } T \textbf{ do } \; c\bar{o}m(\alpha_1) \;||\; \ldots \;||\; \textbf{while } T \textbf{ do } \; c\bar{o}m(\alpha_n)$$

where $c\bar{o}m(\alpha_i)$ is the synchronous communication statement which *matches* $com(\alpha_i)$; i.e. if one of them is a send the other is a receive communication statement. The motivation for introducing this notion is the practical situation where we design a statement (program) by transformations and *after* we have obtained its desired form we proceed to the design of the parallel statements which will communicate with it without deadlock. Then we are never restricted by the future environment of the parallel statement. The inverse situation in communication elimination is similar, extracting the statement from its environment before the simplification and checking for deadlock afterwards.

Two statements S_1 and S_2 are defined to be *congruent*, written as $S_1 \approx S_2$, if $P[S_1]$ and $P[S_2]$ are equivalent for all program contexts $P[S]$. Equivalence of programs means that they have identical sets of *reduced behaviors*, computations retaining an observable part and removing stuttering steps. Two congruent statements are interchangeable in any program. We say that statement S_c *refines* statement S_a, written as $S_c \sqsubseteq S_a$, if for all program contexts $P[S]$, $P[S_c]$ refines $P[S_a]$ (that is, any reduced behavior of $P[S_c]$ is also a reduced behavior of $P[S_a]$). We say that statement S_c is a *flexible refinement* of statement S_a, written as $S_c \sqsubseteq_F S_a$, if for all flexible program contexts $F[S]$, with respect to the external communication operations of S_a and S_c, $F[S_c] \sqsubseteq F[S_a]$.

Any statement in the notation has, implicitly or explicitly, a pre and a post label. It has the familiar sequential composition, also called *concatenation*. Here are some of its laws

Law 1 (Concatenation with Nil) *The following are congruences*

$$\mathbf{nil}; S \approx S \qquad\qquad S; \mathbf{nil} \approx S$$

Nil denotes no transition, no action, in the sense that its pre and post labels are equivalent. A compiler would just delete it.

Law 2 (Concatenation with Skip) *The following are congruences*

$$\mathbf{skip}; S \approx S \qquad\qquad S; \mathbf{skip} \approx S$$

Skip denotes the nul transition, the action which has no effect on memory state variables. However, its pre and post labels are not equivalent. The skip laws only hold in the restricted notation.

Law 3 (Associativity of Sequential Composition) *In general*

$$S_1; \ldots; S_k; \ldots; S_l; \ldots; S_m \;\approx\; S_1; \ldots; [S_k; \ldots; S_l]; \ldots; S_m$$

Parallel composition is also called *cooperation* composition. It has the following laws

Law 4 (Permutation of Cooperation) *Let $p_m(k)$, where $k = 1..m$, denote the* k-th *integer of a permutation of the list $\langle 1, 2, \ldots, m\rangle$. Then:*

$$S_1 || \ldots || S_m \;\approx\; S_{p_m(1)} || \ldots || S_{p_m(m)}$$

Commutativity is a special case. The following laws hold neither in SPL nor in other notations containing synchronous communications.

Law 5 (Cooperation with Nil) *The following are congruences*

$$\mathbf{nil} \;||\; S \approx S \qquad\qquad S \;||\; \mathbf{nil} \approx S$$

Law 6 (Cooperation with Skip) *The following are congruences*

$$\mathbf{skip} \;||\; S \approx S \qquad\qquad S \;||\; \mathbf{skip} \approx S$$

Law 7 (Associativity of Cooperation) *In general*

$$[S_1 || \ldots || S_k || \ldots || S_l || \ldots || S_m\,] \;\approx\; [\,S_1 || \ldots ||\; [\,S_k || \ldots || S_l\,]\;\; || \ldots || S_m\,]$$

The following is a schema of laws defined iteratively and serves for the elimination of a pair of matching synchronous communication statements, which are expressed in SPL as $\alpha \Leftarrow e$ for send and $\alpha \Rightarrow u$ for receive. Where α is a synchronous channel, e is an expression, and u is a variable. All the three are of the same type.

Law 8 (Communication Elimination and Introduction) *In the following all parallel processes are assumed to be disjoint, in the sense that they only read their shared data variables and they do not communicate through asynchronous channels.*

- *Let* T_k^l *and* T_k^r *be statements. Let* H_k^l *and* H_k^r *be statements which do not have communication operations through synchronous channel* α*. In both definitions* $k = 0, 1, \ldots$.

- *Let* $G_0^l = [\alpha \Leftarrow e]$ *and* $G_0^r = [\alpha \Rightarrow u]$ *, and for* $k = 1, 2, \ldots$

  $$G_k^l = H_{k-1}^l; [G_{k-1}^l || P_{k-1}^l]; T_{k-1}^l \quad \text{and} \quad G_k^r = H_{k-1}^r; [G_{k-1}^r || P_{k-1}^r]; T_{k-1}^r$$

  where statements P_k^l *and* P_k^r *can be expressed as* $P_k^l = P_k^{hl}; P_k^{ml}; P_k^{tl}$ *and* $P_k^r = P_k^{hr}; P_k^{mr}; P_k^{tr}$ *, and where any of the* P *processes may be* **nil**, *in such a way that the following holds:*

  Applicability conditions:
  - *None of the* P *statements contain communication statements through channel* α.
  - P_k^{hl} *communicates only with* H_k^r *and* P_k^{hr}.
  - P_k^{hr} *communicates only with* H_k^l *and* P_k^{hl}.
  - P_k^{ml} *communicates only with* G_k*, to be defined below, and* P_k^{mr}.
  - P_k^{mr} *communicates only with* G_k*, to be defined below, and* P_k^{ml}.
  - P_k^{tl} *communicates only with* T_k^r *and* P_k^{tr}.
  - P_k^{tr} *communicates only with* T_k^l *and* P_k^{tl}.

- *Let* $G_0 = [u := e]$ *, and for* $k = 1, 2, \ldots$

  $$G_k = [[H_{k-1}^l; P_{k-1}^{hl}] || [H_{k-1}^r; P_{k-1}^{hr}]]; [G_{k-1} || P_{k-1}^{ml} || P_{k-1}^{mr}];$$

  $$[[P_{k-1}^{tl}; T_{k-1}^l] || [P_{k-1}^{tr}; T_{k-1}^r]]$$

Then $G_n \sqsubseteq_F [G_n^l || G_n^r]$ *for* $n = 1, 2, \ldots$

3 Communication Elimination Tactic

This section introduces a tactic for the application of law 8. We start with some definitions. The following gives the restrictions imposed upon the statement where law 8 and its tactic apply.

Definition 2 (Framework Statement with Communications) *Let* S *be a statement, formed with basic substatements and concatenation, cooperation, iteration and selection substatements. Let* $C : \{\alpha_1, \cdots, \alpha_n\}$ *be the set of synchronous channels whose scope is precisely statement* S*.* S *meets the following requirements:*

1. *All parallel substatements of* S *(processes) are assumed to be disjoint, in the sense that they only read their shared data variables and they do not communicate through asynchronous channels.*

2. *Communication substatements over channels in C are required to be under the scope of neither iteration nor selection substatements.*
3. *Any communication substatement in S over a channel* **not** *in C will not match with any communication substatement in S.*

Statement S will be referred to as the framework statement with internal communications over C *or more simply as* the framework statement.

Definition 3 (Internal Communications) *Channels in C will be referred to as* internal channels of S. *Any communication statement in S over these channels will be referred to as* internal communication substatement of S.

Definition 4 (External Communications) *Any communication statement in S over a channel not in C, referred to as* external channel of S, *will be referred to as an* external communication substatement of S, *and it will not match with any communication statement in S.*

Definition 5 (G statement) *A statement of the form:*

$$G_k^x = H_{k-1}^x; [G_{k-1}^x || P_{k-1}^x]; T_{k-1}^x \quad \textit{for } k = 1, 2, \ldots$$

and where $x = l$ or $x = r$ as defined in law 8 will be called a G statement. *The base case G_0^x is defined in law 8 also.*

Definition 6 (P statement) *A statement of the form:*

$$P_k^x = P_k^{hx}; P_k^{mx}; P_k^{tx} \quad \textit{for } k = 0, 1, \ldots$$

and where $x = l$ or $x = r$ as defined in law 8 will be called a P statement.

Definition 7 (H statement) *A statement which does not have communication statements through synchronous internal channels (channels in C) as defined in law 8 will be called an* H statement.

Definition 8 (T statement) *A tailing statement, as defined in law 8, will be called a* T statement. *It may contain any communication statement.*

The goal of the tactic is to eliminate a matching pair of communication statements from a framework statement S by the application of law 8. Whenever this is not possible no transformation takes place and the boolean variable *done* is set to false.

Our tactic has a preliminary processing, an unification step, and a final transformation. The preliminary processing identifies the key substatements (H, G, P and T) at all the levels k. If necessary *nil* statements are introduced, and the required associations are made so that unification will be possible. The word *compose* will be used in the following to mean those operations done at the preliminary processing. The unification algorithm is a modification of the one given in [MM82]. This amounts to an adaption to our tree-like representation of SPL programs. We use labels within framework statements as a means to identify its

substatements. However the actual tactic uses pointers instead of labels. Hence, as the statement form changes as the tactic evolves, those pointers have to be recalculated.

Given two matching synchronous communication statements $\ell_{G_0^l}$ and $\ell_{G_0^r}$ within some parallelism statement of a framework statement S, the tactic transforms the framework statement into a framework statement S' as a previous step before applying law 8, which refines S' into S'', a framework statement where communication statements $\ell_{G_0^l}$ and $\ell_{G_0^r}$ have been eliminated.

The following is an example for $k = 2$ of the form S' assumed by law 8 which contains the communication statements to be eliminated:

$$\left[\begin{array}{c} H_1^l\,; \\ \left[\begin{array}{c} \left[\begin{array}{c} H_0^l\,; \\ \left[\begin{array}{c} \left[\ell_{G_0^l} : \alpha \Leftarrow e\,;\right] \;||\; \begin{bmatrix} P_0^{hl}\,; \\ P_0^{ml}\,; \\ P_0^{tl} \end{bmatrix} \end{array}\right]\,; \;||\; \begin{bmatrix} P_1^{hl}\,; \\ P_1^{ml}\,; \\ P_1^{tl} \end{bmatrix} \end{array}\right]\,; \\ T_0^l \end{array}\right] \\ T_1^l \end{array}\right] \;||\; \left[\begin{array}{c} H_1^r\,; \\ \left[\begin{array}{c} \left[\begin{array}{c} H_0^r\,; \\ \left[\ell_{G_0^r} : \alpha \Rightarrow u\,;\right] \;||\; \begin{bmatrix} P_0^{hr}\,; \\ P_0^{mr}\,; \\ P_0^{tr} \end{bmatrix} \end{array}\right]\,; \;||\; \begin{bmatrix} P_1^{hr}\,; \\ P_1^{mr}\,; \\ P_1^{tr} \end{bmatrix} \end{array}\right]\,; \\ T_0^r \end{array}\right] \\ T_1^r \end{array}\right]$$

After applying law 8, in this case $k = 2$, we will obtain the following form, corresponding to framework statement S'':

$$\left[\begin{bmatrix} H_1^l \\ P_1^{hl} \end{bmatrix} \;||\; \begin{bmatrix} H_1^r \\ P_1^{hr} \end{bmatrix}\right]\,; \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{bmatrix} H_0^l\,; \\ P_0^{hl} \end{bmatrix} \;||\; \begin{bmatrix} H_0^r\,; \\ P_0^{hr} \end{bmatrix}\right]\,; \\ \left[u := e \;||\; P_0^{ml} \;||\; P_0^{mr}\right]\,; \\ \left[\begin{bmatrix} P_0^{tl}\,; \\ T_0^l \end{bmatrix} \;||\; \begin{bmatrix} P_0^{tr}\,; \\ T_0^r \end{bmatrix}\right] \end{array}\right] \;||\; P_1^{ml} \;||\; P_1^{mr}\,; \left[\begin{bmatrix} P_1^{tl} \\ T_1^l \end{bmatrix} \;||\; \begin{bmatrix} P_1^{tr} \\ T_1^r \end{bmatrix}\right] \end{array}\right]$$

The main algorithm that implements the Communication Elimination Tactic is the following:

```
Communication Elimination Tactic
  inputs   S : framework statement
           ℓ_‖ : label
           ℓ_{G_0^l} : label
           ℓ_{G_0^r} : label
  outputs done : boolean
           S'' : framework statement

  (done, N_level, ℓ_{P_{G_0^l}}, ℓ_{P_{G_0^r}}, WhatCommIsℓ_{G_0^l}) := CollectUnificationInformation(S, ℓ_‖, ℓ_{G_0^l}, ℓ_{G_0^r})
  if (done)
    (done, S') := ComposeGStatement(S, ℓ_{G_0^l}, ℓ_{P_{G_0^l}})
    if (done)
      (done, S') := ComposeGStatement(S', ℓ_{G_0^r}, ℓ_{P_{G_0^r}})
      if (done)
        (done, S') := ComposePStatements(S', ℓ_i, ℓ_j, ℓ_{S_i}, ℓ_{S_j})
```

if $(done)$
  $(done,\ S',\ \ell_{bin}) := \text{ApplyTacticParBinAsso}(S',\ \ell_{\|},\ \ell_{P_{G_0^l}},\ \ell_{P_{G_0^r}})$
  if $(done)$
    $(done,\ S'') := \text{ApplyLaw8}(S',\ \ell_{bin},\ \text{WhatCommIs}\ell_{G_0^l},\ N_{level})$
    if $(done)$
      $(done,\ S'') := \text{ApplyTacticRemoveNils}(S'',\ \ell_{bin},\ N_{level})$
      $(done,\ S'') := \text{ApplyTacticConcatFlat}(S'',\ \ell_{bin},\ N_{level})$

CollectUnificationInformation Starting from a framework statement S, checks whether $\ell_{G_0^l}$ and $\ell_{G_0^r}$ are a matching pair of communication operations. Also determines the level of unification, N_{level}, of law 8 for each of the two top parallel processes that contain statements $\ell_{G_0^l}$ and $\ell_{G_0^r}$, whose labels $\ell_{P_{G_0^l}}$ and $\ell_{P_{G_0^r}}$, respectively, are identified.

ComposeGStatement This procedure is applied for each $\ell_{G_0^x}$, where $x = l$ or $x = r$, and constructs N_{level} G statements, defintion5, so that unification with the left hand side of law 8 with $k = N_{level}$ is possible. This is carried out by transformations applying some of the *nil* or associative laws.

ComposePStatements The P statements of each level (G statement) are determined and composed in this procedure. Whenever required, *nil* statements are introduced and associations of concatenation are made, by law 1 and law 3 to match the applicability conditions of law 8.

ApplyTacticParBinAsso Associates the two top parallel substatements, $\ell_{P_{G_0^l}}$ and $\ell_{P_{G_0^r}}$, of the parallel statement $\ell_{\|}$. This tactic is described in [Bab01]. This is required since law 8 assumes a binary parallelism.

ApplyLaw8 Proper application of law 8 for $k = N_{level}$ at ℓ_{bin}.

ApplyTacticRemoveNils Given the G statements of S'' this tactic removes the *nil*'s introduced in *ComposeGStatement* and *ComposePStatements*, for the unification step.

ApplyTacticConcatFlat Transforms the statement S'' into a concatenation having no concatenation substatement. Therefore, the transformation is recursive in the level of the associations. The associated concatenation substatements have been introduced in *ComposeGStatement* and *ComposeP-Statements*.

We now give the transformation of the following framework statement S, as an illustration of communication statements elimination:

$$S :: \left[\left[\left[\begin{array}{l} H_{10}; \\ \left[\begin{array}{l} H_{11} \\ \| \end{array}\right] \left[\begin{array}{l} H_{12}; \\ \left[\begin{array}{lll} \ell_{G_0^l} : \alpha \Leftarrow e & \| & P_1 \end{array}\right] \end{array}\right] \end{array}\right]\right] \| \left[\begin{array}{l} H_2; \\ \ell_{G_0^r} : \alpha \Rightarrow u; \\ T_2 \end{array}\right]\right]$$

After applying Communication Elimination Tactic we will obtain the following framework statement S'' where the synchronous communication statement over the channel α has been eliminated.

$$S'' :: \left[\begin{array}{l} H_{10}; \\ \left[\begin{array}{l} \left[\begin{array}{lll} H_{12} & \| & H_2 \end{array}\right]; \\ \left[\begin{array}{lll} u := e & \| & P_1 \end{array}\right]; \\ T_2 \end{array}\right] \quad \| \quad H_{11} \end{array}\right]$$

3.1 Collect Unification Information

The algorithm that computes the previous information about the input framework statement S is:

```
Collect Unification Information
  inputs   S : framework statement
           ℓ_‖ : label
           ℓ_{G_0^l} : label
           ℓ_{G_0^r} : label
  outputs done : boolean
           N_level : integer
           ℓ_{P_{G_0^l}}, ℓ_{P_{G_0^r}} : label
           WhatCommIsℓ_{G_0^l} : communication statement flag

  (done, WhatCommIsℓ_{G_0^l}):= MatchConnection(S, ℓ_{G_0^l}, ℓ_{G_0^r})
  if (done)
    (done, N_{level_{ℓ_{G_0^l}}}) := GetNLevel(S, ℓ_{G_0^l}, ℓ_‖)
    if (done)
      (done, N_{level_{ℓ_{G_0^r}}}) := GetNLevel(S, ℓ_{G_0^r}, ℓ_‖)
      if (done)
        N_level:= max(N_{level_{ℓ_{G_0^l}}}, N_{level_{ℓ_{G_0^r}}})
        if (done)
          (done, ℓ_{P_{G_0^l}}):= WhatParProcess(S, ℓ_{G_0^l}, ℓ_‖)
          if (done)
            (done, ℓ_{P_{G_0^r}}):= WhatParProcess(S, ℓ_{G_0^r}, ℓ_‖)
```

MatchConnection Checks whether $\ell_{G_0^l}$ and $\ell_{G_0^r}$ are a matching pair of communication operations, and whether $\ell_{G_0^l}$ is an input or an output communication statement. WhatCommIs$\ell_{G_0^l}$ takes the value *input* if $\ell_{G_0^l}$ is an input communication statement, otherwise takes the value *output*.

GetNLevel Given a $\ell_{G_0^x}$ determines the natural $N_{level_{\ell_{G_0^x}}}$ as the index k, given in law 8, corresponding to the G_k^x statement, where $x = l$ or $x = r$. Different values for $N_{level_{\ell_{G_0^l}}}$ and $N_{level_{\ell_{G_0^r}}}$ are possible.

WhatParProcess If $\ell_{G_0^x} \in \ell_{\|}$ the procedure returns a label $\ell_{P_{G_0^x}}$ that corresponds to a top parallel substatement of $\ell_{\|}$ which $\ell_{G_0^x}$ belongs to, where $x = l$ or $x = r$.

3.2 Compose G Statement

The composition of a G statement is made iteratively by this algorithm:

```
Compose G Statement
  inputs   S : framework statement
           N_level : integer
           ℓ_{G_0} : label
           ℓ_‖ : label
           ℓ_{P_{G_0}} : label
  outputs done : boolean
           S' : framework statement
           ℓ_{P_‖} : label
```

```
S' := S
ℓ_G := ℓ_G0
i_level := 0
while ((i_level ≤ N_level) ∧ done)
  (done, S') := IdentifyGStatement(S, ℓ_G, ℓ_∥, i_level)
  if (ℓ_G∥P of S' is not G statement)
    (done, S') := ApplyLaw5(S', ℓ_G)
    if (done)
      if (ℓ_G∥P of S' is not G∥P statement)
        (done, S') := ApplyLaw4(S', ℓ_G∥P)
        if (done)
          (done, S') := ComposeHTStatements(S', ℓ_G∥P)
          if (done)
            i_level := i_level + 1
            ℓ_G := GetGStatement(i_level)
```

IdentifyGStatement Given a framework statement S the procedure returns a full labeled framework statement S', the new added labels correspond to the H, G, P, T statements for the current level of unification with law 8.

ApplyLaw5 Applies cooperation with nil law, law 5, at ℓ_G. Only when the P statement is missing.

ApplyLaw4 Applies permutation of cooperation law, law 4, at $\ell_{G\|P}$. Only when G is not the left process.

ComposeHTStatements Given the label $\ell_{G\|P}$, the procedure determines the H and T statements which together with the current G $\|$ P statement will form the next level G statement. When necessary *nil*'s are introduced with law 1. Otherwise the associativity law, law 3, is applied.

GetGStatement Returns the label ℓ_G that corresponds to the G statement at the next level i_{level}.

3.3 Compose P Statements

The P statements are identified or formed with *nil* statements, according to the applicability conditions of law 8. Each P statement is composed by $P^x = P^{hx}; P^{mx}; P^{tx}$ where $x = l$ or $x = r$, definition 6.

For example, the algorithm checks whether P^l communicates with any H^r or T^r statements. If it communicates with H^r, then the P statement will be composed as: $P^{hl} = P^l$, and applying law 1 $P^{ml} = nil$ and $P^{tl} = nil$. If it communicates with T^r, then the P statement will be composed as: $P^{tl} = P^l$, and applying law 1 $P^{hl} = nil$ and $P^{ml} = nil$. If P_l does not communicate with H^r and T^r, then the P statement will composed as: $P^{ml} = P^l$, and applying law 1 $P^{hl} = nil$ and $P^{tl} = nil$. Other cases are not treated because they do not satisfy applicability conditions of law 8. A similar algorithm is applied for P^r and H^l and T^l.

4 Elimination of Multiple Pairs

4.1 Introduction

When a framework statement contains more than one communication pair, the tactic given in the previous section has to be applied iteratively for each one of them. This section is devoted to the corresponding tactic.

4.2 Communications within Parallel Statements

This subsection introduces some definitions concerning framework statement S, with its internal and external communications, and the synchronous communication statements, which have to be eliminated from an statement S. In the rest of this section all the substatements of the framework statement S will be referred to more simply as statements.

Definition 9 (Communication Front over Multiple Channels) *Let S be a framework statement. The* communication front of S over the set C of synchronous channels of S, *written* ComFront(C, S), *is defined to be the set of all labels of communication substatements in S over channels in C and which have no other such substatement preceding them in the concatenation ordering of statement S.*

Lemma 1 (Internal Communication Emptyness) *A framework statement S, with set of internal channels C, has no internal communication statements iff* ComFront(C,S)$= \emptyset$.

Definition 10 (Competing Pairs of a Communication Front) *Let S be a framework statement, C the set of its internal channels, and* ComFront(C,S) *its communication front. Then, the* set of its competing pairs, *written* CompPairs(C,S) *is, by definition*

$$\{(l, m) | l, m \in \text{ComFront}(C, S) \wedge l \textit{ matches } m\}$$

i.e the set of all possible matching pairs formed with statements in ComFront(C,S).

When a matching pair is eliminated, communication statements which where not in the communication front may be *uncovered*, in the sense that they belong now to the new front.

Definition 11 (Uncovered Set of Communication Statements) *Let S be a framework statement, (l, m) be a pair of matching communication statements in* ComFront(C,S), *and S' be the statement which results after the elimination of this pair from S. Then, the* set of communication statements uncovered by (l, m) *in S, written* UncovSet((l,m),S), *is formed by the communication statements which are in* ComFront(C,S') *but not in* ComFront(C,S)

4.3 The Iterative Tactic

Given a framework statement S this tactic applies recursively the tactic *Communication Elimination* that applies law 8. The set $CompPairs(C, S, \ell_{\|})$, defined above, is computed at each iteration. To apply law 8 we need at least one matching pair of communication statement. In order to provide some detail the following algorithm describes this tactic:

```
Elimination of Multiple Pair
  inputs   S : framework statement
  outputs done : boolean
           S' : framework statement

  do
    C_set := SetOfSynchronousChannels(S)
    CF_set := ComFront(C_set, S)
    if (CF_set ≠ ∅)
      CP_set := CompPairs(CF_set, S)
      if (CP_set ≠ ∅)
        (ℓ_{G_0^l}, ℓ_{G_0^r}) := GetSet(CP_set)
        ℓ_|| := ParallelStatement(S, ℓ_{G_0^l}, ℓ_{G_0^r})
        (done, S') := ApplyTacticComEli(S, ℓ_||, ℓ_{G_0^l}, ℓ_{G_0^r})
  until (CF_set = ∅ ∨ CP_set = ∅)
```

Before applying the tactic *Communication Elimination* the algorithm computes the set $CompPairs$ to determine where to apply law 8. C_{set} is the set of all internal synchronous channels of S and CF_{set} is the communication front over C_{set}. The set of competing pairs is computed if there exists any communication statement in CF_{set}, otherwise the tactic terminates. The final step in the iteration body applies the proper communication elimination tactic, detailed in the previous section, to the pair obtained by *GetSet* and to the parallelism statement $\ell_{\|}$ embodying the pair and determined by $ParallelStatement$. If CP_{set} is empty the tactic terminates also. The algorithm returns a boolean value that indicates whether it has succeed or not, and the transformed framework statement S' only in the case of success.

5 Conclusions

Multiple synchronous communication statements of imperative concurrent programs can be eliminated by applying the iterative tactic presented in this paper. This tactic always applies a sequence of laws taken from a well defined set, then we can guarantee that the transformation of the program carried out by the tactic preserves its properties.

The tactic only applies to a subset of programs which we refer to as framework statements, and which have to be expressed in a restricted version of the SPL notation. The justification of this has been presented in related works, and it is needed so that the laws used in the communication elimination tactic hold.

The determination of the P statements in different ways is open. We have presented only one of the multiple possible partitions and composition of the P

statements. Criteria such as the preservation of the maximum parallelism of the H, G, P and T statements can be applied also.

As a future work we could extend the tactics to more general programs. The synchronous communication statements, which have to be eliminated from a framework statement S, could be allowed to appear as guards of communication selection substatements and they would be eliminated also by the iterative tactic.

References

[Bab01] F. Babot, M. Bertran, A. Climent, M. Nicolau, Some Tactics for Communication Parallelism and Elimination, in Actas IX Jornadas de Concurrencia, Siges, June 2001, pp. 1-12.

[BaWr98] R-J. Back, J. von Wright, *Refinement Calculus. A Systematic Introduction.* Springer-Verlag 1998.

[Bens00] S. Bensalem, M. Bozga, J.C. Fernandez, L. Ghirvu, Y. Lakhnech, *A Transformational Approach for Generating Non-Linear Invariants.* In J. Palsberg (Ed.), *Static Analysis*, Proc. 7th Intl. Symp. SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000. LNCS Vol. 1824, Springer, 2000, pp. 58-74.

[Ber01] M. Bertran, F. Babot, A. Climent, M. Nicolau, Communication and Parallelism Introduction and Elimination in Imperative Concurrent Programs, in *Static Analysis*, LNCS v.2126, Springer, 2001, pp 20-39.

[Ber97] M. Bertran, F. Alvarez-Cuevas, A. Duran, Communication Extended Abstract Types in the Refinement of Parallel Communicating Processes, in *Transformation-Based Reactive Systems Development*, LNCS v.1231, Springer, 1997.

[BBC+00] N.S. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H.B. Sipma, and T.E. Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 16, 227-270, June 2000.

[BBC+95] N.S. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[Broo96] S.D. Brookes. 'Full abstraction for a shared variable parallel language', *Information and Computation*, 127(2):145-163, June 1996.

[Broy93] M. Broy, 'Functional Specification of Time-Sensitive Communicating Systems', *ACM Transactions on Software Engineering and Methodology*, 2(1),: 1-46, January 1993.

[Broy97] M. Broy, 'Refinement of Time', in M. Bertran and T. Rus (eds.), *Transformation-Based Reactive Systems Development*, Springer-Verlag, Lecture Notes in Computer Science 1231, 1997, pp. 44-63.

[Broy99] M. Broy, 'A Logical Basis for Component-Based Systems Engineering', *Tech. Report Inst. fr Informatik*, Tech. Univ. Munchen, Germany.

[ChaMi88] K.M. Chandy and J. Misra, *Parallel Program Design*, Addison Wesley, 1988.

[Chin00] Wei-Ngan Chin, Sian-Cheng Khoo, Z. Hu, M. Takeidu, *Deriving Parallel Codes via Invariants.* In J. Palsberg (Ed.), *Static Analysis*, Proc. 7th Intl. Symp. SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000. LNCS Vol. 1824, Springer, 2000, pp. 75-94.

[Cla99] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking*, The MIT Press, 1999.

[JDin98] J. Dingel, 'A Trace-Based Refinement Calculus for Shared-Variable Parallel Programs', in A.Martin Haeberer (Ed.) *Algebraic Methodology and Software Technology*, AMAST'98, LNCS 1548, Springer-Verlag, pp. 231-247, 1998.

[FiMaSi98] B. Finkbeiner, Z. Manna, H. Sipma, *Deductive Verification of Modular Systems.* In *Compositionality: The Significant Difference*, COMPOS'97, LNCS v. 1536, pp. 239-275, Springer 1998.

[GrMlWa79] M. Gordon, A.J. Milner, Ch. P. Wadsworth, *Edinburgh LCF*, LNCS v.78, Springer-Verlag, 1969.

[Hoa78] C.A.R. Hoare, 'Communicating Sequential Processes', *Communications of ACM*, Vol 21, pp 666-677, 1978.

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[Holt91] Gerald Holtzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[Hoo94] J. Hooman, 'Extending Hoare Logic to Real-Time', *Formal Aspects of Computing*, 6A: 801-825, BCS, 1994.

[KeMaPn94] Y. Kesten, Z. Manna, A. Pnueli, 'Temporal Verification of Simulation and Refinement', In REX Symposium *A Decade of Concurrency*, Lecture Notes in Computer Science 803, pp. 273-346, Springer-Verlag, 1994.

[Lam94] L. Lamport, 'The Temporal Logic of Actions', *ACM Trans. Progr. Lang. and Sys.*, 16(3):872-923.

[BMah94] B. Mahony, 'Using the Refinement Calculus for Dataflow Processes'. *Tech. Report 94-32*, Soft.Verification Research Centre, University of Queensland, October 94.

[MaPn91] Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems. Specification.* Springer-Verlag, 1991.

[MaPn94] Z. Manna, A. Pnueli, *Temporal Verification of Reactive Systems. Safety.* Springer-Verlag, 1995.

[MM82] Aberto Martinelli and Ugo Montanari, *An Efficient Unification Algorithm*, ACM Transaction on Programming Languages and Systems (TOPLAS) **4** (1982), no. 2, 258–282.

[McMil93] K.L. McMillan, and D.L. Dill, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic, 1993.

[Mil80] R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag, 1980.

[Mil89] R. Milner, *Communication and Concurrency*, Prentice-Hall 1989.

[Mull99] M.Muller-Olm, D.A. Schmit, B. Steffen, *Model Checking: A Tutorial Introduction.* In A.Cortesi, G.File (Eds.), *Static Analysis*, Proc. 6th Intl. Symp. SAS'99, Venice, Italy, September 22-24, 1999. LNCS, Vol 1694, Springer, 1999, pp. 330-354.

[Ngan00] Wei-Ngan Chin, Sian-Cheng Khoo, Z. Hu, M. Takeidu, *Deriving Parallel Codes via Invariants.* In J. Palsberg (Ed.), *Static Analysis*, Proc. 7th Intl. Symp. SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000. LNCS Vol. 1824, Springer, 2000, pp. 75-94.

[Podl00] A. Podelski, *Model Checking as Constraint Solving.* In J. Palsberg (Ed.), *Static Analysis*, Proc. 7th Intl. Symp. SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000. LNCS Vol. 1824, Springer, 2000, pp. 22-37.

PACOBIR: Un prototipo de un sistema CBIR distribuido

M. Luisa Córdoba[1], Oscar D. Robles[2], Angel Rodríguez[2], M. Isabel García[1], María S. Pérez[1], Manuel Nieto[1], Antonio Pérez[1], Luis Pastor[3], and José L. Bosque[3]

[1] DATSI-Universidad Politécnica de Madrid, Campus de Montegancedo s/n, 28660 Boadilla del Monte (Madrid), España,
{mcordoba,mgarcia,mperez,mnieto,aperez}@fi.upm.es,
WWW home page: http://www.datsi.fi.upm.es

[2] DTF-Universidad Politécnica de Madrid, Campus de Montegancedo s/n, 28660 Boadilla del Monte (Madrid), España,
{orobles,arodri}@dtf.fi.upm.es,
WWW home page: http://www.dtf.fi.upm.es

[3] Escuela Superior de Ciencias Experimentales y Tecnológicas, URJC. C. Tulipán, s/n. 28933 Móstoles. Madrid. Spain.
{lpastor,j.l.bosque}@escet.urjc.es

Resumen Este artículo presenta una implementación paralela de un sistema CBIR. La aplicación se ha diseñado para un entorno multiprocesador de memoria distribuida y se ha implementado de forma experimental en un *cluster* de PCs. Las principales características de la implementación aquí descrita son su buena escalabilidad, que permite al prototipo incorporar nuevas entradas dentro del sistema CBIR, y su buena relación precio/prestaciones. La solución propuesta permite implementar este sistema sobre otras arquitecturas paralelas minimizando el coste de la migración y manteniendo los niveles de eficiencia alcanzados en la configuración analizada.

1 Introducción

El nivel de madurez logrado en áreas tan diferentes como el procesado de señal, la visión computacional, las bases de datos o la interacción hombre-máquina, junto con la pujante evolución de los sistemas de comunicaciones, ha traído consigo la proliferación de sistemas de información cuyo objetivo es la gestión y almacenamiento eficiente de grandes volúmenes de datos multimedia. La aparición de este tipo de sistemas ha generado la necesidad de emplear técnicas alternativas para acceder a los datos almacenados, en lugar de utilizar las técnicas clásicas de recuperación de datos incorporadas a las bases de datos convencionales. Este tipo de sistemas se centra en la búsqueda de datos en función del contenido real de los mismos, dando lugar a los llamados sistemas CBIR[1] [17,3,15]. En general,

[1] *Content Based Information Retrieval Systems.*

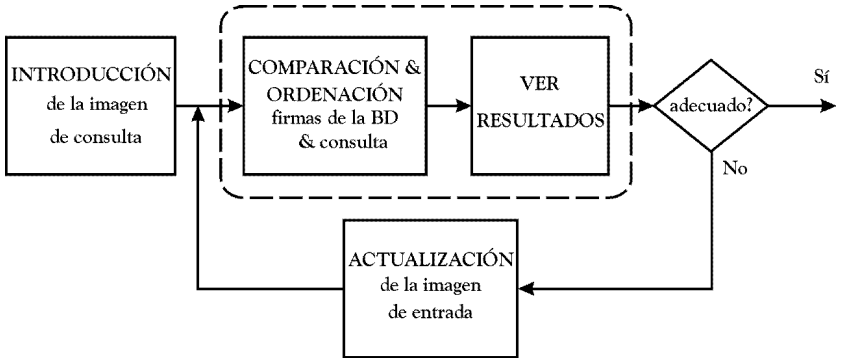


Figura 1. Funcionamiento de PACOBIR.

el volumen de los datos almacenados es muy significativo, como por ejemplo sucede con las bases de datos de logotipos, y puede ser necesario recurrir a implementaciones paralelas para almacenar y acceder a los datos disponibles de forma eficiente [13,18].

Este trabajo se centra en la descripción de una implementación distribuida de un sistema CBIR de imágenes 2D en color. Las principales características de dicha implementación son su buena escalabilidad, flexibilidad, portabilidad y su buen ratio coste/prestaciones.

El contenido de este artículo se puede descomponer en los siguientes apartados. La sección 2 presenta una panorámica general de la implementación de PACOBIR[2]. Las herramientas gráficas para visualización y realización de búsquedas se describen en la sección 3. En la sección 4 se detallan las primitivas utilizadas en las búsquedas. Las secciones 5 y 6 presentan, respectivamente, ciertos detalles sobre la base de datos utilizada en los experimentos y la implementación CBIR paralela. Finalmente, en la sección 7 se presentan lo resultados experimentales, mientras que las principales conclusiones extraídas se encuentran en la sección 8.

2 Visión general de PACOBIR

PACOBIR es un prototipo de un sistema CBIR distribuido que realiza la recuperación de la información mediante el contenido completamente visual de las imágenes 2D de color almacenadas en una base de datos. La herramienta es plenamente operativa y flexible, permitiendo al usuario realizar sobre la base de datos todas las operaciones típicas de este tipo de sistemas: búsquedas, inserciones, supresiones, etc. La búsqueda se puede descomponer en las siguientes etapas (Figura 1):

[2] *PArallel COntent Based Image Retrieval.*

1. Introducción de la imagen de entrada/búsqueda y cálculo de la firma[3].
2. Comparación de las firmas de la imagen de entrada y de las de la BD y ordenación de los resultados.
3. Visualización de los resultados.
4. Actualización de la imagen de entrada si los resultados no son satisfactorios y vuelta al paso 1.

Este esquema refleja una arquitectura compuesta por los siguientes subsistemas:

- Un sistema gráfico de búsquedas y visualización que proporcione un interfaz amigable que permita la realización de búsquedas visuales (Sec. 3).
- Un procedimiento de extracción de las características más relevantes de las imágenes de entrada en color y que calcule una métrica de semejanza entre la imagen de entrada y las imágenes almacenadas en el sistema (Sec. 4).
- Un sistema de base de datos que permita la gestión de grandes conjuntos de datos de diferente naturaleza: textual, numérica o gráfica (Sec. 5).
- Un sistema de comunicación que intercambie datos entre los diferentes nodos del sistema CBIR para obtener una implementación eficiente (Sec. 6).

El almacenamiento de todas las imágenes del sistema se convierte en un problema cuando el tamaño del sistema CBIR crece demasiado, haciendo necesaria la utilización de mecanismos de almacenamiento que permitan la distribución de las imágenes entre diferentes dispositivos. En la implementación descrita en este artículo, se distribuyen tanto los cálculos como las imágenes entre los n nodos del sistema. Este enfoque es muy eficiente y reduce las necesidades de almacenamiento de cada nodo.

Un análisis detallado de las operaciones involucradas revela que hay una etapa que demanda una carga computacional mucho mayor que las demás: la etapa de comparación y ordenación, en la que todas las firmas del sistema se contrastan con la firma de la imagen de entrada con el fin de seleccionar las más similares. Afortunadamente, no hay ni dependencias de datos, ni algorítmicas, por lo que la explotación del paralelismo de datos se puede hacer dividiendo la carga de trabajo entre los n nodos independientes (etapas en línea punteada en la Fig. 1).

3 Interfaz de usuario

Una de las claves del éxito de un sistema CBIR es la utilización de una herramienta gráfica para realizar las consultas y visualizar los resultados. Dicha herramienta debe permitir una interacción amistosa entre el usuario y el sistema CBIR, así como una *realimentación relevante*[4] [3] de los resultados obtenidos en una consulta. La herramienta desarrollada para conseguir este objetivo en PACOBIR es un interfaz gráfico de usuario (GUI) que permite acceder a la base

[3] La firma es un vector de características extraído de la imagen; representa simbólicamente su semántica.

[4] *Relevance feedback.*

de datos a usuarios inexpertos en este área, y por lo tanto, sin conocimientos de la sintaxis de las consultas. Además, ofrece la posibilidad de realizar consultas directas a través de un intérprete de comandos para usuarios experimentados.

A continuación se citan algunas de las operaciones más importantes que se pueden realizar mediante el GUI del prototipo: selección e inserción de imágenes en el sistema CBIR así como la descripción del contenido de dichas imágenes mediante información textual, como por ejemplo los códigos de Viena en bases de datos de logotipos [16]; diferentes tipos de búsquedas: firmas semejantes o firmas semejantes en combinación con información textual parecida (i.e. los códigos de Viena en las bases de datos de logotipos); definición de distintas métricas de semejanza para cada firma; configuración del número de nodos entre los que está distribuida la base de datos; incorporación de nuevas firmas a las imágenes del sistema CBIR; configuración de la zona de visualización para establecer el número y el tamaño de las imágenes que se presentan al usuario; cambio de la base de datos a utilizar en el sistema CBIR.

El GUI se ha implementado utilizando Glade, una herramienta de libre distribución para la construcción de interfaces desarrollada dentro del proyecto GNOME. Glade permite la construcción interactiva de interfaces de usuario gráficas GNOME/GTK. El interfaz generado se puede salvar en un fichero XML y traducid ir directamente a módulos en lenguaje C. Dichos módulos contienen la definición de los *widgets* y de las funciones (manejadores) asociadas a los distintos eventos que se pueden producir en el interfaz. El usuario define el nombre de dichas funciones y Glade genera la estructura del código, que debe rellenar el usuario.

El motivo para la elección de esta herramienta es que garantiza un corto período de desarrollo y asegura la portabilidad a otras plataformas, así como la reutilización de los módulos implementados. La Figura 2 muestra el aspecto del interfaz desarrollado.

4 Primitivas CBIR y proceso de cálculo de semejanza

Las firmas implementadas en PACOBIR se basan en la transformada de Haar de cada uno de los canales RGB de los datos de entrada. Los coeficientes de la transformada *wavelet* muestran variaciones en las características de los objetos a diferentes niveles de resolución. Los coeficientes de detalle de la transformada *wavelet* se pueden considerar como un proceso de extracción de las componentes de alta frecuencia de los objetos que aparecen en las imágenes, mientras que los coeficientes de análisis se comportan de forma complementaria: cuanto más bajo es el nivel de resolución, generan regiones más homogéneas. Se han implementado varias primitivas basadas en el color y en la forma. Este enfoque permite al usuario seleccionar las primitivas de acuerdo al tipo de búsqueda que quiere hacer. Las primitivas implementadas son las siguientes [11,10]:

- Basadas en el color:
  - Energía de los coeficientes de la transformada de Haar.
  - Histogramas globales multirresolución de color.

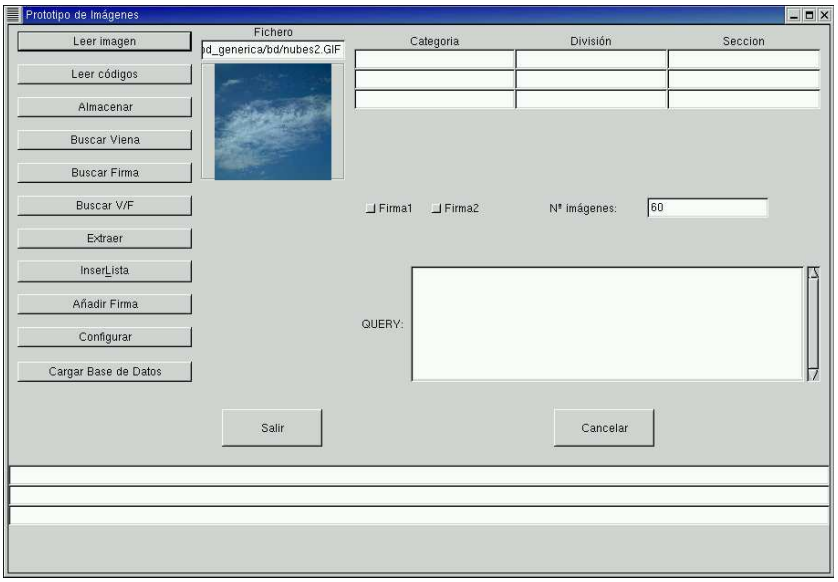


Figura 2. Interfaz de usuario de la herramienta de consulta/visualización.

- Basadas en la forma:
  - Invariantes de forma, como por ejemplo los momentos de Hu y Zernike [5,14].
  - Número de esquinas extraídas de la transformada de Haar.

Para fusionar información procedente de diferentes primitivas se han probado dos alternativas:

1. Concatenar las firmas individuales de cada primitiva, normalizando el vector resultante para conseguir dominios de representación homogéneos.
2. Trabajar de forma independiente con cada primitiva y seleccionar después aquella en la que el objeto buscado, si se conoce, consigue el mejor resultado.

Aunque la segunda opción tiene un mayor coste computacional, se puede implementar con cierta facilidad una versión paralela de ambas técnicas [1]. El clasificador utilizado en la etapa de comparación es el de mínima distancia [11].

5 Gestor de la BD

Un sistema que gestiona y procesa en paralelo un gran volumen de datos, debe paralelizar tanto el procesamiento de los datos, como su sistema de almacenamiento para evitar cuellos de botella [4]. Por ello, en PACOBIR se utiliza un subsistema multidisco para almacenar las bases de datos diseñado de acuerdo a dos criterios principales: versatilidad y eficiencia. Versatilidad para adaptarse fácilmente a los cambios en el sistema CBIR y eficiencia para conseguir tiempos

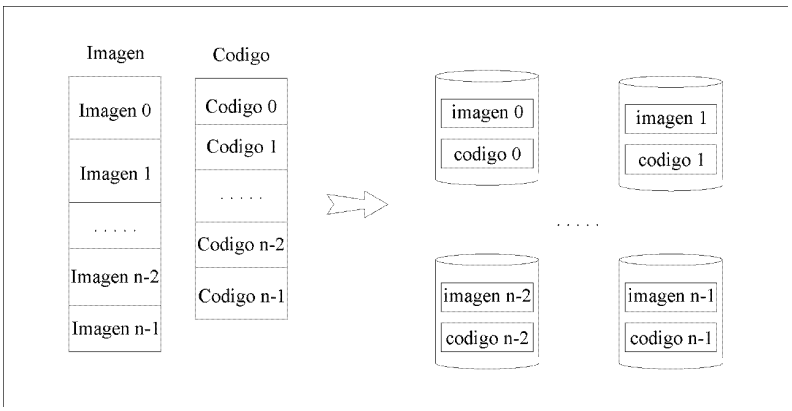


Figura 3. Distribución de la información sobre los nodos de PACOBIR.

de respuesta bajos aunque se manejen grandes volúmenes de datos. Teniendo en cuenta estas dos premisas, se han diseñado dos entidades que representan el contenido de cada elemento incluido en la base de datos. La primera entidad, `imagen`, contiene la información gráfica de las imágenes; es decir, nombre, ubicación, firmas, etc. La segunda, `codigo`, contiene descriptores textuales del contenido de las imágenes, como por ejemplo los Códigos de Viena usados en la clasificación de logotipos.

Para realizar la paralelización del sistema de almacenamiento se ha elegido una base de datos secuencial de libre distribución, MYSQL [7]. Incluso si hubiera disponible un servidor de bases de datos paralelo de libre distribución que fuera completo, fiable y eficiente, no sería la mejor solución para nuestro sistema, ya que no tendría la flexibilidad requerida y realizaría automáticamente muchas operaciones de sincronización, control, etc. que son innecesarias en PACOBIR, lo que conllevaría una pérdida de eficiencia notable [8,2].

La paralelización de la base de datos se ha realizado teniendo en cuenta los siguientes aspectos: el particionamiento de las bases de datos, la evitación de interdependencias en los datos, y el equilibrio de carga.

Para distribuir las bases de datos entre los discos disponibles, se ha elegido un esquema de *particionamiento horizontal* [2,9] según el cual las entidades `imagen` y `codigo` se dividen en porciones y se almacenan en los discos de tal forma que cada porción contiene la información de un subconjunto de las imágenes almacenadas en la base de datos (Fig. 3). Este esquema de particionamiento permite que toda la información correspondiente a cada imagen sea almacenada en el mismo disco y que todas las imágenes almacenadas en cada disco sean gestionadas de forma independiente por una instancia de MYSQL y sin interdependencias con los datos gestionados por las otras instancias de MYSQL. De esta forma, el paralelismo ofrecido por el *cluster* de PCs puede ser explotado completamente.

PACOBIR lleva a cabo un equilibrado de carga estático, ya que la migración de imágenes de un disco a otro es una operación costosa que no se puede realizar

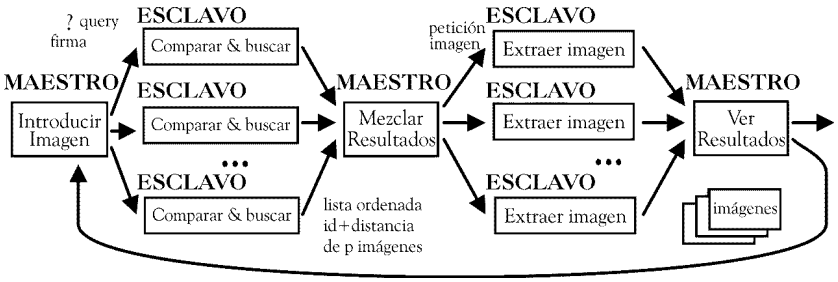


Figura 4. Comunicación entre procesos en la solución paralela.

mientras hay usuarios operando con las bases de datos. Sin embargo, se pueden seguir varias alternativas. En primer lugar, la distribución de las imágenes puede plantearse teniendo en cuenta los tamaños de los discos cuando se utiliza un *cluster* heterogéneo. También se puede efectuar de acuerdo con el rendimiento de cada nodo buscando obtener tiempos de respuesta similares y, como resultado, conseguir el máximo rendimiento global del sistema. Finalmente, la distribución de las imágenes se puede hacer siguiendo criterios organizativos ajenos al prototipo.

La naturaleza paralela de PACOBIR no es un problema significativo a la hora de mantener la integridad de las bases de datos. Solamente hace falta un control adicional a la hora de añadir imágenes en una base de datos para evitar la inserción en un nodo de una imagen que ya esté almacenada en otro nodo, o la inserción simultánea de una misma imagen en dos nodos por parte de usuarios distintos. Este control se puede efectuar de la forma más simple y eficiente, ya que, una vez que el sistema CBIR ha sido creado, la adición de nuevas imágenes es una operación relativamente infrecuente que consume muy pocos recursos y una insignificante cantidad de tiempo. Todos los controles necesarios para mantener la integridad de los datos y la coherencia de la información se realizan localmente mediante las instancias de MYSQL en la forma usual de los sistemas no paralelos, ya que gestionan toda la información relativa a las imágenes almacenadas en sus discos.

6 Implementación paralela utilizando MPI

El paradigma de programación paralela que mejor se adapta al problema es el de granja de procesos cooperantes: un proceso *maestro* reparte el trabajo entre otros procesos *esclavos*, y cuando éstos han acabado, el proceso *maestro* recoge los resultados parciales obtenidos por cada uno de los procesos *esclavos*. Esta solución es adecuada gracias a las escasas dependencias entre datos y algorítmica, ya que la sobrecarga de comunicación derivada del problema es muy pequeña. La Figura 4 muestra un diagrama de comunicaciones entre los procesos del sistema.

La aplicación está basada en el modelo de programación de paso de mensajes y utiliza librerías MPI como primitivas de comunicación entre el proceso *maestro*

y los *esclavos*. Se ha utilizado MPI porque actualmente constituye un estándar en la programación basada en paso de mensajes para arquitecturas paralelas, y además ofrece una buena portabilidad entre distintas arquitecturas. Hemos utilizado la versión de libre distribución 6.5.2 de LAM [12], del Laboratory for Scientific Computing de la Universidad de Notre Dame. Las principales características de esta versión de MPI son la incorporación de las nuevas funcionalidades definidas en la especificación MPI-2 [6] y su disponibilidad para un amplio grupo de plataformas diferentes.

La comunicación por paso de mensajes implica enviar por la red las imágenes encontradas, así como cierta información adicional de control de la transmisión entre origen y destino. Esta comunicación extra debe ser mínima y lo más compacta posible. Cuando se encuentra una imagen en la base de datos, ciertos datos acerca de esa imagen se almacenan en una estructura `Imagen_encontrada`:

```
typedef struct Imagen_encontrada
{
    char id_nodo[TAM_NOMBRES];
    char dir[TAM_NOMBRES];
    char fich[TAM_NOMBRES];
    int contador;
    double distancia;
} Imagen_encontrada;
```

El significado de los campos de esta estructura es el siguiente:

id_nodo: identificador del nodo donde está la imagen.
dir: directorio o *path* en el nodo donde está la imagen.
fich: nombre del fichero de la imagen encontrada.
contador: `distancia` entre el campo de texto de la imagen encontrada y el de la imagen de referencia. Por ejemplo, para una base de datos de logotipos, este campo podría implementarse como una métrica basada en la clasificación de códigos de Viena.
distancia: valor de semejanza respecto a la firma seleccionada. Este valor puede estar basado en un clasificador de mínima `distancia` [11].

En cada nodo *esclavo*, con la información de las p imágenes más parecidas a la de referencia, se forma un *array* de p elementos (cada uno de ellos es una estructura `Imagen_encontrada`), al que denominamos `Imagenes_encontradas`. Antes de enviar la estructura `Imagenes_encontradas` al *maestro*, el *esclavo* las ordena, eligiendo como criterio de ordenación los campos `distancia` y `contador`, si es que este último se ha definido y utilizado como tal. En una primera etapa, esta estructura de datos con cierta información de las imágenes encontradas es la única información que los *esclavos* envían al *maestro* (ver Fig. 4). Después, el *maestro* reúne toda la información de los *esclavos* en un vector global donde cada elemento es a su vez un *array* del tipo de datos `Imagenes_encontradas`, donde está la información que le envió cada *esclavo*. Este vector se denomina `Imagenes_encontradas_global`. Una vez recopilados los arrays de todos los

Proceso 1 Query–Maestro

Calcular la firma de la imagen de entrada {Inicializar la conexión con la base de datos}
Construir una tira de caracteres con la consulta (tipo de función de búsqueda, etc...)
for *rank* = 1 a *ntareas* − 1 **do** {Enviar la consulta a cada *esclavo*}
  MPI_Send(query, TAM_QUERY, MPI_CHAR, rank, SEARCH_TAG, MPI_COMM_WORLD);
end for
for *rank* = 1 a *ntareas* − 1 **do** {Recibir el número de imágenes encontradas seguido del vector de estructuras `Imagenes_encontradas` de cada *esclavo*. Almacenar la información de `Imagenes_encontradas` enviadas por los *esclavos* en el vector `Imagenes_encontradas_global`.}
  MPI_Recv(&i, 1, MPI_INT, rank, SEARCH_TAG, MPI_COMM_WORLD, &status);
  MPI_Recv(&(Imagenes_encontradas_global[rank-1]), 1, vector_MPI, rank, SEARCH_TAG, MPI_COMM_WORLD, &status);
end for
Mezclar todas las imágenes encontradas según el criterio de ordenación utilizado en cada *esclavo*.

esclavos, el *maestro* mezcla los registros de todas las imágenes ordenadamente, siguiendo el mismo criterio que siguieron a su vez los *esclavos*. Por último, el *maestro* pide a los *esclavos* las p imágenes cuyos campos `distancia` y/o `contador` son los p mejores en la etapa de mezcla. Cuando ha recibido todas las imágenes, las compone formando un mosaico que presenta al usuario.

La sobrecarga de comunicación introducida es despreciable respecto al tiempo de respuesta del proceso completo de la consulta del usuario. No obstante, se han definido dos estructuras de datos específicas de MPI, `MPI_Type_struc` y `MPI_hvector`, para enviar los datos por la red sin tener que empaquetar variables, ya que esta solución tiene mejor rendimiento que la alternativa del empaquetado/desempaquetado añadido al envío de cualquier mensaje.

El pseudo-código correspondiente a la implementación de la consulta de los procesos *maestro* y *esclavo* se puede apreciar en los Procesos 1 y 2.

Es posible que los distintos nodos hagan accesos concurrentes, pudiéndose ejecutar simultáneamente ambos procesos, *maestro* y *esclavo*, en cada nodo del *cluster*.

7 Resultados experimentales

En la implementación de PACOBIR se diferencian claramente dos fases: una primera en la que se implementaron y comprobaron individualmente los cuatro subsistemas descritos en las secciones anteriores, y una segunda fase en la que el principal problema fue integrar estas cuatro herramientas.

La configuración del sistema descrito es un *cluster* de cuatro PCs conectados por una red Ethernet a 100 Mb/s. Cada nodo de proceso dispone de los siguientes

Proceso 2 Query–Esclavo

MPI_Recv(query, TAM_QUERY, MPI_CHAR, 0, SEARCH_TAG, MPI_COMM_WORLD, &status); {Recibir una tira de caracteres con la consulta del *maestro*} {Inicializar la conexión con la base de datos}
Ejecutar localmente la consulta con la parte de la base de datos asignada a este *esclavo*.
Para cada imagen encontrada, completar los campos de la estructura `Imagenes_encontradas`.
MPI_Send(&i, 1, MPI_INT, 0, SEARCH_TAG, MPI_COMM_WORLD); {Devolver al *maestro* el número de imágenes encontradas, y el vector `Imagenes_encontradas` con la información de las p imágenes más semejantes seleccionadas en este *esclavo*.}
MPI_Send(global_found_images, 1, vector_MPI, 0, SEARCH_TAG, MPI_COMM_WORLD);

componentes: un procesador AMD Athlon a 650 MHz, memoria principal de 256 MB y memoria cache de 512 KB, y un disco duro SCSI con capacidad de 9 GB. El sistema operativo instalado en los PCs es Linux v. 2.2.14. La Figura 5 muestra el resultado del sistema ante la consulta de la Figura 2.

La solución adoptada es bastante escalable, como se demostró en Bosque *et al.*[1], donde se probó una implementación paralela similar al sistema CBIR aquí descrito formada por un *cluster* de veinticinco PCs y un conjunto de datos compuesto por 29,5 millones de firmas y 1,455 Tb. En ese experimento se obtuvo una eficiencia alrededor del 97% y un *speedup* casi lineal, aunque el sistema descrito en este trabajo tenía una funcionalidad bastante más limitada desde el punto de vista del usuario.

Aunque PACOBIR pone a disposición del usuario la utilización de varias firmas, tal y como se ha descrito en la sección 4, sólo se presentan aquí los resultados obtenidos con algunas de ellas, que se han considerado representativas para nuestros objetivos. Las primitivas que hemos seleccionado son los códigos de Viena, la energía de los coeficientes multirresolución de la transformada de Haar, (etiquetada como Firma 1 en los sucesivo), y los histogramas globales multirresolución en color (etiquetada como Firma 2).

Los experimentos de medida se han centrado en el tiempo de búsqueda, ya que es esta etapa la que se beneficia de las optimizaciones y de la paralelización, por ser la que más recursos necesita, como se mencionó en la sección 2. Hemos elegido dos parámetros para evaluar el rendimiento del sistema con las tres primitivas seleccionadas: el número de nodos utilizados en la paralelización y el tamaño de la base de datos sobre la que se hacen los experimentos. El tamaño de la base de datos se puede expresar también como el número de imágenes entre las que se hace la búsqueda, y así se ha considerado en la notación de las Tablas y Figuras. Los valores que en ellas aparecen son en realidad valores medios de varias medidas realizadas para cada una de las imágenes de muestra utilizadas.

En la Tabla 1 existe una relación lineal entre el tamaño de la base de datos y el número de nodos del *cluster*. El número total de imágenes consideradas

Figura 5. Resultado de la búsqueda con la imagen nubes21 en el sistema PACOBIR.

Tabla1. Tiempos medios de búsqueda (s) al aumentar el tamaño de la base de datos, que mantiene una relación lineal con el número de nodos del *cluster*.

	Secuencial	Par. 1 *esclavo*	Par. 2 *esclavos*	Par. 3 *esclavos*
No. de imágenes	4254	4254	8380	12646
Cód. Viena .	0,016	0,570	0,581	0,587
Firma 1	12,912	13,192	13,637	13,961
Firma 2	17,008	17,074	17,632	17,996

en la base de datos es de 4.254 imágenes para la versión secuencial y para la versión paralela con un sólo nodo, 8.380 imágenes para la de dos *esclavos*, y 12.646 imágenes para la de tres *esclavos*. Cada nodo (*esclavo*) almacena en su disco su parte asignada de la base de datos (ver Sección 5). En la Figura 6 se puede apreciar que el tiempo de búsqueda mantiene una tendencia prácticamente constante al aumentar el tamaño de la base de datos.

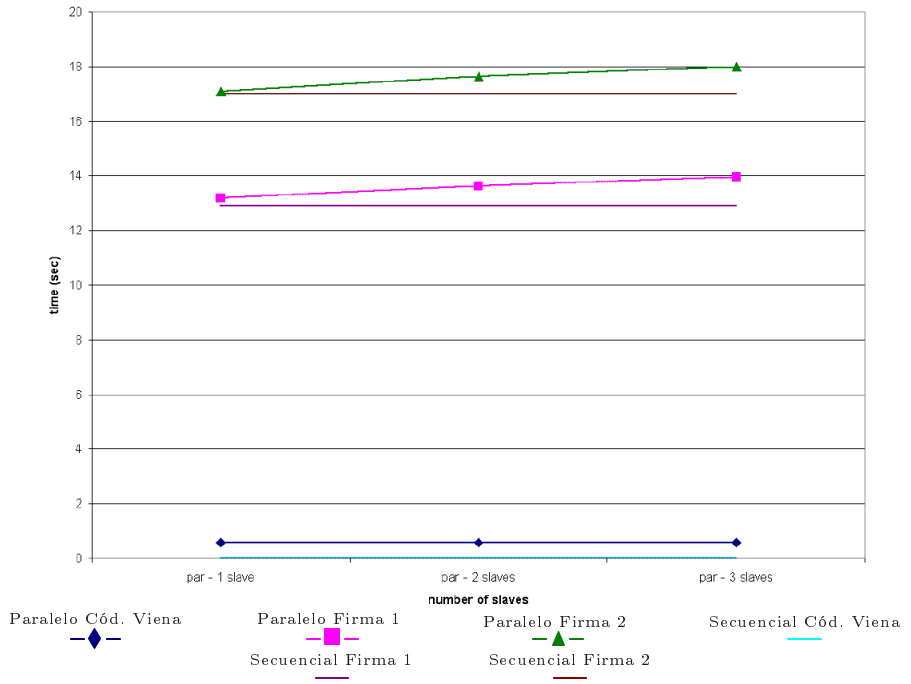


Figura 6. Tiempos medios de búsqueda (s) al aumentar el tamaño de la base de datos (sólo para el caso paralelo).

Tabla2. Tiempos medios de búsqueda (s) para el mismo tamaño de base de datos (12.646 imágenes).

	Secuencial	Par. 1 *esclavo*	Par. 2 *esclavos*	Par. 3 *esclavos*
Cód. Viena	0.100	0.586	0.670	0.698
Firma 1	52.562	53.323	22.076	13.864
Firma 2	60.194	60.963	26.981	17.930

En la Tabla 1 se puede constatar que, aunque la búsqueda para la firma 1 consume menos tiempo que para la firma 2, ambas firmas sufren una evolución similar al aumentar el número de imágenes a tratar.

La Tabla 2 muestra medidas de tiempo equivalentes a las de la Tabla 1, pero con el mismo número de imágenes en todas las configuraciones. En este caso, se han generado tres bases de datos diferentes: una con las 12.646 imágenes en un solo nodo, una segunda en la que se repartieron las 12.646 imágenes en dos nodos (*esclavos*), y una tercera base de datos con un tercio de las 12.646 imágenes almacenado en cada uno de los tres *esclavos*. Al igual que en la Figura 6, la Figura 7 muestra gráficamente la evolución del tiempo de búsqueda comparando varias configuraciones del *cluster* con el caso secuencial. En ambas figuras, el tiempo

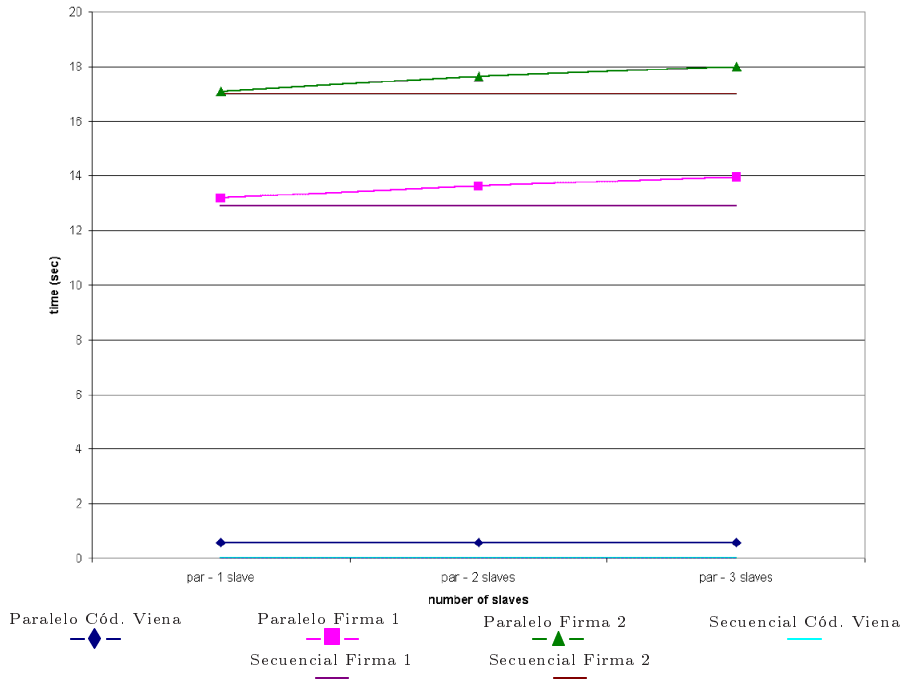


Figura 7. Tiempos medios de búsqueda (s) para el mismo tamaño de la base de datos.

de búsqueda para el caso secuencial se ha representado como una línea de valor constante como referencia para compararla con los tiempos de los casos paralelos. En la Figura 7, el tiempo secuencial se midió con una base de datos local (4.245 imágenes), mientras que en la Figura 6 el tiempo secuencial se midió con toda la base de datos almacenada en un único nodo (12.646 imágenes).

La sobrecarga de comunicación debida a las primitivas MPI de la versión paralela es prácticamente despreciable, como puede verse en la Figura 7, que compara el tiempo de respuesta de la versión secuencial y la versión paralela con un solo *esclavo*. Se puede observar en las Tablas 1 y 2 que en el caso paralelo el tiempo de búsqueda basada en los códigos de Viena se degrada más que en el caso de utilizar otras firmas. Esto se debe a que la búsqueda por campos de texto consume mucho menos tiempo que la basada en firmas. Cuanto más tiempo consume el proceso de búsqueda, mayor es el beneficio que se puede esperar de la paralelización y distribución. En la Tabla 2, cabe destacar la relación superlineal entre el tiempo de búsqueda y el número de *esclavos* involucrados en los experimentos. Esto puede ser debido fundamentalmente a dos razones:

– La gestión interna de las tuplas de la base de datos en el proceso de búsqueda depende del número de tuplas almacenadas en cada *esclavo*.

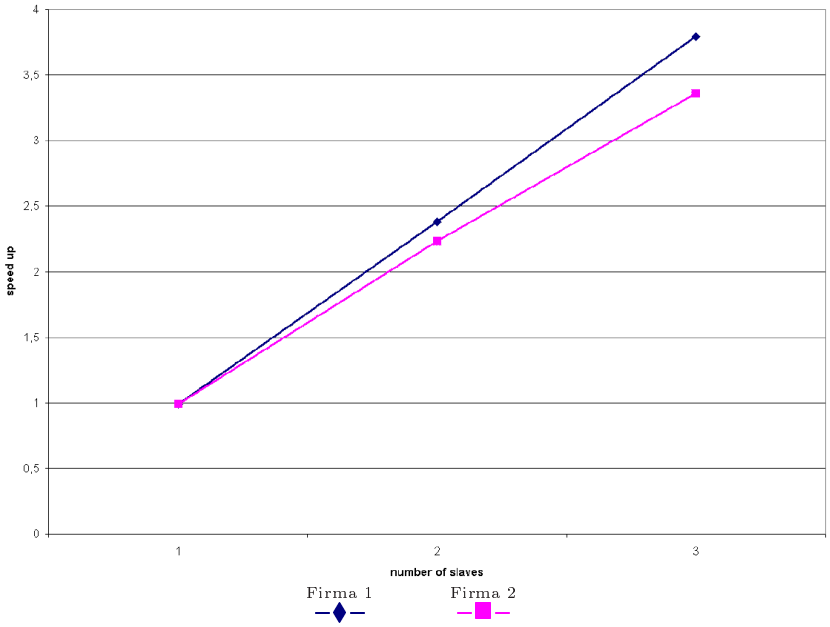


Figura 8. Speedup real

– El tamaño de la memoria cache del sistema global aumenta proporcionalmente al número de *esclavos*, por lo que el tiempo medio de acceso a memoria varía en las distintas configuraciones.

La Figura 8 muestra el *speedup* obtenido a partir de las Tablas 1 y 2 para la Firma 1 y la Firma 2. Por último, la Figura 9 muestra la eficiencia obtenida en la paralelización.

8 Conclusiones y futuras líneas de trabajo

Se ha presentado una implementación distribuida de un sistema CBIR. El sistema soporta búsqueda por contenido visual de imágenes 2D en color almacenadas en una base de datos. La herramienta integra cuatro subsistemas: una interfaz gráfica de usuario, un módulo de extracción automática y emparejado de características gráficas, un sistema de gestión de bases de datos, y un esquema de comunicación paralelo. El prototipo es funcional y flexible, permitiendo al usuario realizar las operaciones típicas sobre cualquier base de datos.

Un aspecto a destacar del sistema CBIR implementado es su bajo coste de comunicación, lo que deriva en una paralelización con unos resultados muy satisfactorios. La solución propuesta, basada en un *cluster* de cuatro PCs que utilizan MPI, ofrece al usuario tiempos de respuesta muy pequeños. Esto es debido a las escasas dependencias algorítmica y de datos, así como a la escasa

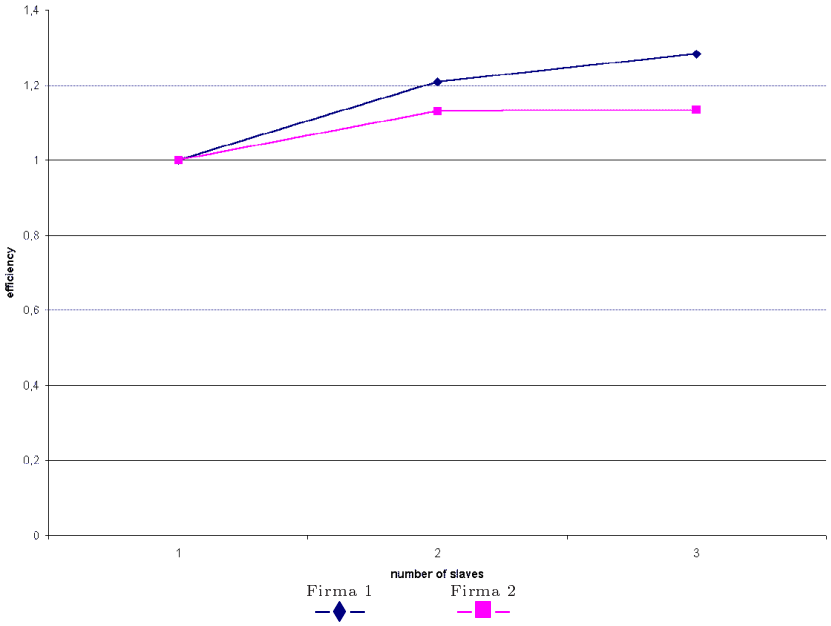


Figura 9. Eficiencia

sobrecarga debida a la comunicación. Otros aspectos importantes de la solución basada en el *cluster* es su buena relación coste/prestaciones y la escalabilidad del sistema. Estas características hacen especialmente adecuada esta solución cuando se tiene que manejar un volumen de datos muy elevado. Además, la utilización del estándar MPI proporciona *portabilidad* al sistema.

El trabajo futuro está encaminado a introducir estrategias para el balance dinámico de la carga del *cluster*, lo que facilitará su implementación en *cluster* heterogéneos. Otro aspecto a tener en cuenta en las próximas implementaciones es la incorporación de mecanismos de *tolerancia a fallos*.

Agradecimientos

Este trabajo ha sido parcialmente subvencionado por la CICYT (proy. CICYT TIC98-0272-C02-01 y TIC99-0947-C02-01) y por la Comunidad Autónoma de Madrid (proy. 07T/0014/2001 y 07T/0008/2001).

Referencias

1. José Luis Bosque, Oscar David Robles, Angel Rodríguez, and Luis Pastor. Study of a parallel CBIR implementation using MPI. In Virginio Cantoni and Concettina Guerra, editors, *Proceedings on International Workshop on Computer Architectures for Machine Perception, IEEE CAMP 2000*, pages 195–204, Padova, Italy, September 2000. ISBN 0-7695-0740-9.

2. P. de Miguel, M. Nieto, et al. Operating systems and programming environments for parallel computers. Technical Report WP3-APD-1-9106, ESPRIT Project 2528 - Supernode, June 1992.
3. Alberto del Bimbo. *Visual Information Retrieval*. Morgan Kaufmann Publishers, San Francisco, California, 1999.
4. D. Dewitt and J. Gray. Parallel database systems: The future of high performane database systems. *Communications of the ACM*, 35(5):85–98, June 1992.
5. M. Hu. Visual pattern recognition by moment invariants. *IEEE Trans. Info.Theory*, 8:179–187, 1962.
6. MPI Forum. A Message-Passing Interface standard, 2001. www.mpi-forum.org
7. MySQLAB. MySQL. Web, 2001. http://www.mysql.com/
8. M. Nieto, P. de Miguel, J. Carretero, and D. Bernabé. Multidisk X/OPEN ISAM implementation of a RDMS. Technical Report 67.1/ARQ/92, Facultad de Informática de Madrid, Campus de Montegancedo s/n, 28660 Boadilla del Monte (Madrid), Spain, November 1991.
9. M. Nieto, P. de Miguel, J. Carretero, and C. Corbacho. A multidisk sql server for general purpose shared-nothing multiprocessors. In *4th Working Group Workshop on Massively Parallel Computing*, pages 72–82, Madeira (Portugal), April 1993. Proj. COST 229.
10. Oscar D. Robles, Angel Rodríguez, and M. Luisa Córdoba. A study about multiresolution primitives for content-based image retrieval using wavelets. In M. H. Hamza, editor, *IASTED International Conference On Visualization, Imaging, and Image Processing (VIIP 2001)*, pages 506–511, Marbella, Spain, September 2001. IASTED, ACTA Press. ISBN: 0-88986-309-1.
11. Angel Rodríguez, Oscar D. Robles, and Luis Pastor. New features for Content-Based Image Retrieval using wavelets. In Fernando Muge and Rogério Caldas Pinto and Moisés Piedade, editors, *V Ibero-american Simposium on Pattern Recognition, SIARP 2000*, pages 517–528, Lisbon, Portugal, September 2000. ISBN 972-97711-1-1.
12. Jeffrey M. Squyres, Kinis L. Meyer, M. D. McNally, and Andrew Lumsdaine. *LAM/MPI User Guide*. University of Notre Dame, 1998. LAM 6.3.
13. S. Srakaew, N. Alexandridis, P. Piamsa Nga, and G. Blankenship. Content-based multimedia data retrieval on cluster system environment. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *High-Performance Computing and Networking. 7^{th} International Conference, HPCN Europe 1999*, pages 1235–1241. Springer Verlag, 1999.
14. M. R. Teague. Image analysis via the general thoery of moments. *J. Opt. Soc. Am.*, 70(8):920–929, August 1980.
15. Colin C. Venters and Matthew Cooper. A review of content-based image retrieval systems. Technical report, Manchester Visualization Center. Manchester Computing. University of Manchester, July 2000.
16. World Intellectual Property Organization. Vienna classification. Web, 2001. Color image database.
17. Minerva M. Yeung, Boon-Lock Yeo, and Charles A. Bouman, editors. *Proceedings of the SPIE/EI'2000 Symposium on Storage and Retrieval for Media Databases*, volume 3972. SPIE, January 2000. ISBN 0-8194-3590-2.
18. Mohamed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, October 1999.

On the use of formal models in Software Performance Evaluation*

Juan Pablo López-Grao, José Merseguer, and Javier Campos

Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain,
{jpablo,jmerse,jcampos}@posta.unizar.es

Abstract. Importance of performance evaluation at first stages of the software development life-cycle has been progressively rising. We believe that the need for integration of formal models in the software engineering process is a must in order to apply well-known analysis techniques to software models. In previous papers it has been stated our proposal of extension of UML semantics for some diagram types and a complete method to translate them into GSPN models. Here we will focus on activity diagrams: a new translation method for them will be presented, while we explain their link with other UML diagrams such as statecharts so as to amplify the expressivity at system description. Last but not least, our CASE tool prototype will be introduced. As it will be seen, every modeling aspect for these diagrams will be covered and, thanks to it, the translation process will be automatically performed.

Keywords: UML, software performance, Generalized Stochastic Petri nets, compositionality, modeling, CASE tool

1 Introduction

Unified Modeling Language (UML) [7] is a semi formal language developed by the Object Management Group [7] to specify, visualize and document models of software systems and non-software systems too. UML defines three categories of diagrams: static diagrams, behavioural diagrams and diagrams to organize and manage application modules. Behavioural diagrams are intended to describe system dynamics, therefore they play a prominent role for us since the objective of our works is the performance evaluation [20] of software systems at the first stages of the software development process [23, 25]. These diagrams belong to five kinds: Use Case diagram, Sequence diagram, Activity diagram, Collaboration diagram and Statechart diagram.

Our proposal consists in introducing new syntactical elements in UML diagrams to model performance concepts. By doing so, the software engineer can model behavioural, functional and performance requirements in a consistent fashion. In this paper the role played by the Activity diagram (AD) for the

* This work has been developed within the project P084/2001 of the Gobierno de Aragón, and the project UZ00-TEC-03 of the Universidad de Zaragoza.

performance evaluation of software systems is fully analyzed under the perspective of this proposal [19]. Since UML diagrams are not meant for performance evaluation and moreover its semantics is "informally" defined, we translate them into Generalized Stochastic Petri nets (GSPN) [1], gaining a formal semantics for them and besides an analyzable model. Obviously, the translation implies taking decisions on the interpretation of the diagrams.

So far we have dealt with the Sequence diagram (SD) (by means of the UML collaborations package) and the Statechart diagram (SC) [6] (by means of the UML state machines package). In the case of the AD we base our interpretation on the fact that ADs are suitable for internal flow process modeling, therefore they are relevant to describe activities performed by the system, usually expressed in the SC as doActivities in the states.

In this paper we investigate the key concepts to describe performance issues in the context of the AD and we give a formal semantics for the AD in terms of GSPN, compatible with that proposed for the SD and the SC in [6]. Furthermore, we briefly overview our prototype tool, which implements both aspects: its GUI is designed as a front end to model annotated ADs whereas the tool itself constructs their translation into GSPNs in order to analyze them with the GreatSPN tool [12].

We adopt the notation defined in [1] for GSPNs, but simplified to consider only ordinary systems (Petri nets in which arcs have weight at most one). A GSPN system is a 8-ple $S = (P, T, \Pi, I, O, H, W, M^0)$, where P is the set of places, T is the set of immediate and timed transitions, $P \cap T = \emptyset$; $\Pi : T \longrightarrow \mathbb{N}$ is the priority function that maps transitions onto natural numbers representing their priority level, by default, timed transitions have priority equal to zero; $I, O, H : T \longrightarrow 2^P$ are the input, output, inhibition functions, respectively, that map transitions onto the power set of P; $W : T \longrightarrow \mathbb{R}$ is the weight function that assigns real (positive) numbers to rates of timed transitions and to weights of immediate transitions. Finally, $M^0 : P \longrightarrow \mathbb{N}$ is the initial marking function.

A labeled ordinary GSPN (LGSPN) is then a triplet $\mathcal{LS} = (S, \psi, \lambda)$, where S is a GSPN ordinary system, as defined above, $\lambda : T \longrightarrow L^T \cup \tau$ is the labeling function that assigns to a transition a label belonging to the set $L^T \cup \tau$ and $\psi : P \longrightarrow L^P \cup \tau$ is the labeling function that assigns to a place a label belonging to the set $L^P \cup \tau$. τ-labeled net objects are considered to be internal.

Note that, with respect to the definition of LGSPN system given in [10], here both places and transitions can be labeled, moreover, the same label can be assigned to place(s) and to transition(s) since it is not required that L^T and L^P are disjoint.

The rest of the article is organized as follows: Section 2 describes the proposed annotations for the ADs and enumerates the main rules of the translation method. Section 3 analyzes the translation of each element in the AD into a stochastic Petri net model. Section 4 discusses how the stochastic Petri net model for the whole AD is obtained. Section 5 briefly presents our tool prototype. Finally, section 6 summarizes the paper, explores the bibliography and discusses future extensions.

2 Activity Diagrams for Performance Evaluation

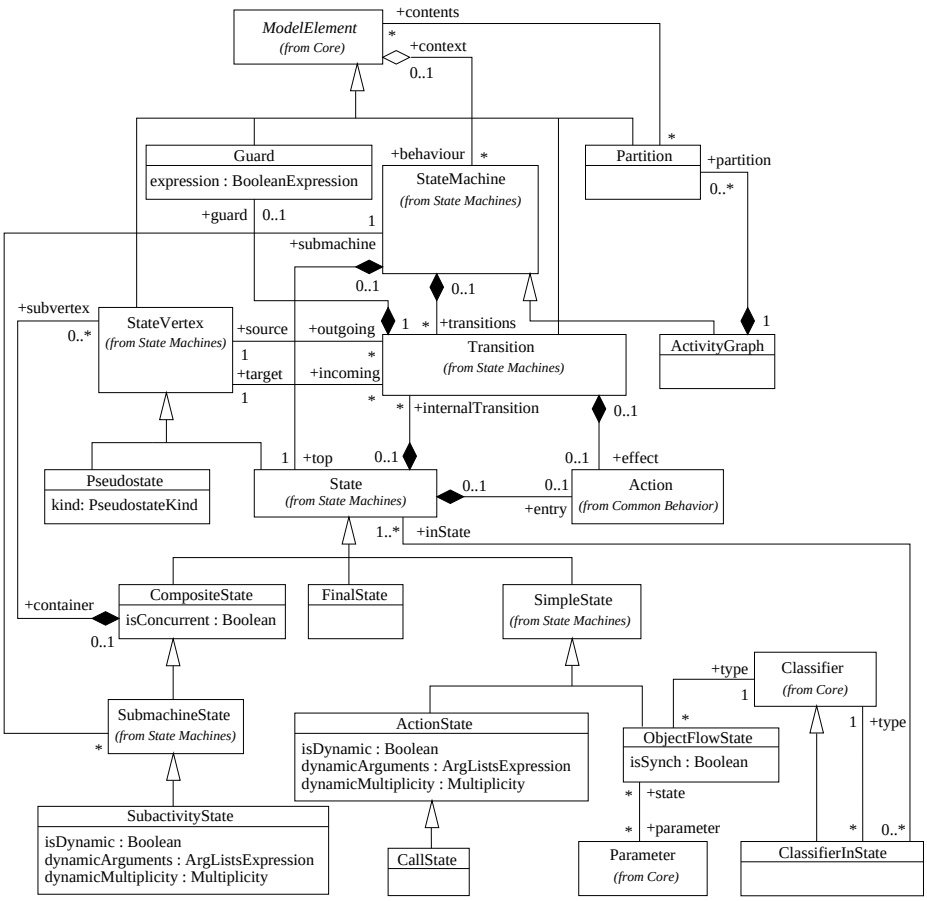


Fig. 1. UML Activity Graphs metamodel (extended)

Activity Diagrams represent UML activity graphs and are just a variant of UML state machines (see [7], section 3.84). In fact, a UML activity graph is a specialization of a UML state machine (SM), as it is expressed in the UML metamodel (see figure 1). The main goal of ADs is to stress the internal control flow of a process in contrast to statechart diagrams, which represent UML SMs and are often driven by external events.

As our objective is to use ADs to obtain performance measures of the model element they describe, we need additional modeling information, such as routing rates or the duration of the basic actions. We propose to annotate the AD to collect this information: subsection 2.1 describes this proposal.

It must be noted that in this paper we only focus in those elements proper of ADs. See that, according to UML specification ([7], section 3.84), almost every state in an AD should be an action or subactivity state, so almost every transition[1] should be triggered by the ending of the execution of the entry action or activity associated to the state. Anyway, UML is not strict at this point, so elements from state machines package could occasionally be used.

As far as it is concerned, our decision is not to allow other states than action, subactivity or call states, and thus to accept only the use of external events by means of call states and control icons involving signals, i.e. signal sendings and signal receipts. As a result of this, events are always deferred (as any event is always deferred in an action state), so an activity will not ever be interrupted when it is described by an AD. Further attempts to include other SM elements are not discarded and could be object of future work, although they introduce some new problems.

Anyway, we suggest the use of SCs to describe the dynamical behaviour of those parts of the system dependable of external events.

2.1 Performance annotations

Our proposal is to include two different aspects in our annotations: time and probability. As it was stated in previous papers [19], we will use tagged values as an extensibility mechanism to integrate them in our UML models. Annotations will be attached to transitions instead of states as in previous works, in order to allow the assignment of different action durations depending on the decision. Anyway, any other syntax is accepted as long as it is consistent with the process described below.

It must be noticed that, in the following, we will use the notion of not-timed transitions in the scope of ADs to specify those arcs which have no time annotation or to which a duration equal to zero is assigned. Doing so, we are trying to avoid confusions with immediate transitions, as long as they are different concepts in the domain of UML SMs.

The format suggested is {n sec.; P(k)} or {n-m sec.; P(k)} for timed transitions and {P(k)} for not-timed transitions. If no probability P(k) is provided we will assume an equiprobable sample space, i.e., identical probability for each 'brother' transition to be triggered. As it is shown, we allow time expressed in terms of an estimated value or a range of them. We have discarded the usage of packet size annotations as proposed for SMs [19] due to the fact that ADs are commonly used to model internal control flow. Figure 5 shows some examples of annotations (in braces).

Time annotations will be allocated wherever an action is executed (outgoing transitions of such states, including outgoing transitions of decision pseudostates with an action state as input) and probability annotations wherever a decision

[1] Notice that the word 'transition' has different meanings in UML and PNs domain. We preserve both meanings in this paper as the context should be enough to discriminate the 'transition' we are referring to (UML or PN 'transition')

is taken, i.e. next to guard conditions. It must be noticed that there is a special case where the performance annotation is attached to the state instead of the outgoing transition: when the control flow is not shown because it is implicit in the action-object flow. We do so because we do not want to have performance annotations applied to it, as it has a different semantics.

2.2 Translation rules and formal definitions

A brief description of each AD element and their translation to LGSPNs is presented in the next section. Section 4 illustrates the method to compose those LGSPNs to obtain the whole model for a concrete AD according to our proposed semantics. We must note that, in the following, we suppose that every object derived from ModelElement metaclass has an unique name within its namespace, although it could be not explicitly shown in the model.

As a rule, the translation of each one of AD elements can be summarized as a three-phased process:

step 1 Translation of each outgoing and self-loop transition. Applicable to action, subactivity and call states, and to fork pseudostates. Depending on the kind of transition, a different rule must be applied (see figures 2 and 4).

step 2 Composition of the LGSPNs corresponding to the whole set of each kind of transitions considered in step 1. Applicable to action, subactivity and call states, and to fork pseudostates.

step 3 Working out the LGSPN for the element by superposition of the LGSPNs obtained in the last step (if any) and, occasionally, an additional LGSPN corresponding to the entry to its associated state.

The formal definition of one of the LGSPN systems shown in Figure 2 is stated below. The rest of the cases in Figures 2 and 4 are straightforward from this example, so they will not be explicitly illustrated.

From now onward, we will adopt the Object Constraint Language [7] (OCL) syntax to indicate the image of an element (or of a set of elements) belonging to the domain of a certain relation. Let us suppose there exists an association between the classes D and C and let *rel* be the name of the role played by the class C in the association, then the image of an instance d of class D, through the derived relation *rel*: $D \rightarrow C$, is denoted as $d.rel$. Also the attributes of a class D, say at_1, and at_2 are denoted using the dot notation, $D.at_1$, and $D.at_2$.

A system for an outgoing timed transition *ott* of an action state *AS* (see figure 2, case 1.a) is a LGSPN $\mathcal{LS}^{ott}_{AS} = (S^{ott}_{AS}, \psi^{ott}_{AS}, \lambda^{ott}_{AS})$ characterized by the set of transitions $T^{ott}_{AS} = \{t_1, t_2\}$, and the set of places $P^{ott}_{AS} = \{p_1, p_2, p_3\}$. The input and output functions are respectively equal to:

$$I^{ott}_{AS}(t) = \begin{cases} \{p_1\} & \text{if } t = t_1 \\ \{p_2\} & \text{if } t = t_2 \end{cases} \qquad O^{ott}_{AS}(t) = \begin{cases} \{p_2\} & \text{if } t = t_1 \\ \{p_3\} & \text{if } t = t_3 \end{cases}$$

There are no inhibitor arcs, so $H^{ott}_{AS}(t) = \emptyset$. The priority and the weight functions are respectively equal to:

$$\Pi_{AS}^{ott}(t) = \begin{cases} 0 & \text{if } t = t_2 \\ 1 & \text{if } t = t_1 \end{cases} \qquad W_{AS}^{ott}(t) = \begin{cases} r_{ott} & \text{if } \Pi_{AS}^{ott}(t) = 0 \\ p_{cond} & \text{if } \lambda_{AS}^{ott}(t) = cond\_ev \\ 1 & \text{otherwise} \end{cases}$$

where, in this case, r_{ott} is the rate parameter of the timed transition t_2 and p_{cond} is the weight of the immediate transition t_1.

The weight p_{cond} is assigned the value of the probability annotation attached to the AD transition *ott*. If there is not such annotation, p_{cond} is equal to $1/nt$, where nt is the number of elements in the set $AS.outgoing$.

The rate r_{ott} is equal to $1/n$ if the time annotation attached to the AD transition is expressed in the format $\{n\ sec.\}$, or equal to $2/n+m$ if it is expressed in the format $\{n-m\ sec.\}$. Furthermore, in our CASE tool (presented in section 5) the last case is considered as a parameter of the system in order to automatize the analysis of the final LGSPN for different values within the range specified.

The initial marking function is defined as $\forall p \in P_{AS}^{ott} :\ M_{AS}^{ott0}(p) = \emptyset$. Finally, the labeling functions are equal to:

$$\psi_{AS}^{ott}(p) = \begin{cases} ini\_AS & \text{if } p = p_1 \\ execute & \text{if } p = p_2 \\ ini\_nextx & \text{if } p = p_3 \end{cases} \qquad \lambda_{AS}^{ott}(t) = \begin{cases} cond\_ev & \text{if } t = t_1 \\ out\_\lambda & \text{if } t = t_2 \end{cases}$$

where, for abuse of notation, $AS = AS.name$ and $nextx = ott.target.name$.

As they are profusely used in next section, we also define AG as the activity diagram, $Lstvertex^P$ the set of labels of state vertices in it, $Lstvertex^P = \{ini\_target, \forall target \in AG.transitions \rightarrow target.name\}$ and Lev^P as the set of events in the system, $Lev^P = \{e\_evx, \forall evx \in Ev\} \cup \{ack\_evx, \forall evx \in Ev\}$.

3 Translating Activity Diagram elements

The following subsections are devoted to translate each diagram element into a LGSPN; the composition of these nets (section 4) results in a stochastic Petri net system that will be used to obtain performance parameters for the modelled element.

3.1 Action states

An action state is 'a shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action' ([7], section 3.85). According to this definition and the translation of simple states in SMs [19] we should interpret the action atomic and therefore represent it by an immediate transition within the LGSPN corresponding to the state. However, if we considered every action immediate (for action states), then most of the activities modelled by ADs would be immediate too, when they are expected to have a concrete duration. So we will distinguish between timed and not-timed transitions (in ADs) to determine the type of transition needed —timed or immediate— and its rate associated in the resulting LGSPN.

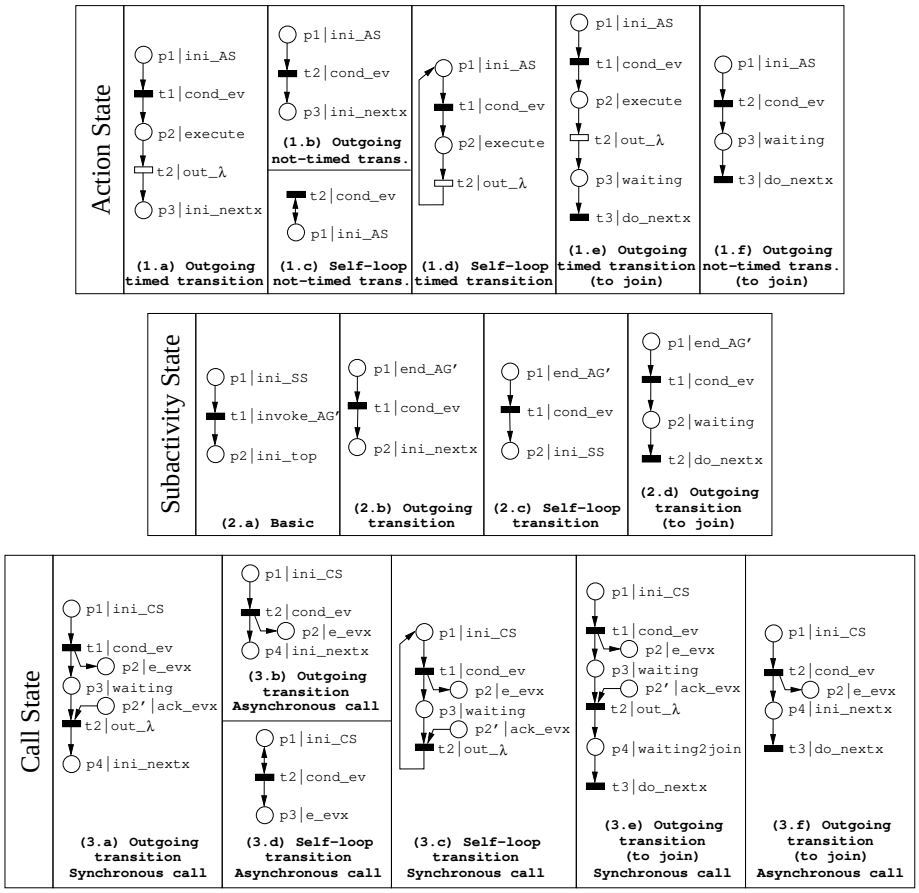


Fig. 2. Action, Subactivity and Call States to LGSPN

Translating an action state into LGSPN formalism takes the three steps expressed in section 2.2. Given an action state AS let q be the number of outgoing timed transitions OT_i of the state (which do not end in a join pseudostate), q' the number of outgoing not-timed transitions ON_j (which do not end in a join pseudostate), r the number of outgoing timed transitions OTJ_m that end in a join pseudostate, r' the number of outgoing not-timed transitions OTN_n that end in a join pseudostate, s the number of self-loop timed transitions ST_k and s' the number of self-loop not-timed transitions SN_l.

Then for each outgoing or self-loop transition t, we have a LGSPN $\mathcal{LS}^t_{AS} = (S^t_{AS}, \psi^t_{AS}, \lambda^t_{AS})$ as shown in figure 2, cases 1.a-1.f. This results in a set of $q + q' + r + r' + s + s'$ LGSPN models that need to be combined to get a model of the state AS, $\mathcal{LS}_{AS} = (S_{AS}, \psi_{AS}, \lambda_{AS})$.

Firstly we must compose the submodels of the transitions of the same type, using the superposition operators defined in Appendix A and the following equations:

$$\mathcal{LS}_{AS}^{OT} = \underset{Lstvertex^P}{\overset{i=1,\ldots,q}{||}} \mathcal{LS}_{AS}^{OT_i} \qquad \mathcal{LS}_{AS}^{ON} = \underset{Lstvertex^P}{\overset{j=1,\ldots,q'}{||}} \mathcal{LS}_{AS}^{ON_j}$$

$$\mathcal{LS}_{AS}^{ST} = \underset{ini\_AS}{\overset{k=1,\ldots,s}{||}} \mathcal{LS}_{AS}^{ST_k} \qquad \mathcal{LS}_{AS}^{SN} = \underset{ini\_AS,out\_\lambda}{\overset{l=1,\ldots,s'}{\bigsqcup}} \mathcal{LS}_{AS}^{SN_l}$$

$$\mathcal{LS}_{AS}^{OTJ} = \underset{ini\_AS}{\overset{m=1,\ldots,r}{||}} \mathcal{LS}_{AS}^{OTJ_m} \qquad \mathcal{LS}_{AS}^{ONJ} = \underset{ini\_AS}{\overset{n=1,\ldots,r'}{||}} \mathcal{LS}_{AS}^{ONJ_n}$$

Again composing the subsystems just shown, the LGSPN model $\mathcal{LS}_{AS}$ is now defined by:

$$\mathcal{LS}_{AS} = ((((\mathcal{LS}_{AS}^{SN} \underset{ini\_AS}{||} \mathcal{LS}_{AS}^{ST}) \underset{ini\_AS}{||} \mathcal{LS}_{AS}^{ON}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AS}^{OT}) \underset{ini\_AS}{||} \mathcal{LS}_{AS}^{OTJ}) \underset{ini\_AS}{||} \mathcal{LS}_{AS}^{ONJ}$$

Finally we must remember that UML lets any kind of action to be executed inside an action state. That means we might find a CallAction or a SendAction there. However, UML syntax provides two concrete elements for this type of states: call states and signal sending icons. We suggest their use, but if an action state is used instead, then we should apply the translation method described for the equivalent element (call state or signal sending control icon).

3.2 Subactivity states

A subactivity state always invokes a nested AD. Its outgoing transitions do not have time annotations attached, as the duration activity can be determined translating the AD and composing the whole system (that will be seen later in this paper).

Translating a subactivity state into LGSPN formalism takes the three steps expressed in section 2.2. Notice that there is an additional LGSPN that corresponds with the entry to the state, called *basic*.

Then, given a subactivity state SS let q be the number of outgoing transitions O_i of the state (which do not end in a join pseudostate), r the number of outgoing transitions OJ_k that end in a join pseudostate, and s the number of self-loop transitions S_j. Also let AG' be the nested activity diagram and *top* the name of the first element of AG', $top = AG'.top$.

According to the translations shown in figure 2, cases 2.a-2.d, we have a basic LSGPN $\mathcal{LS}_{SS}^{B} = (S_{SS}^{B}, \psi_{SS}^{B}, \lambda_{SS}^{B})$ and one LGSPN for each outgoing or self-loop transition t, $\mathcal{LS}_{SS}^{t} = (S_{SS}^{t}, \psi_{SS}^{t}, \lambda_{SS}^{t})$. Therefore, we have $q + r + s + 1$ LGSPN models that need to be combined to get a model of the state SS, $\mathcal{LS}_{SS} = (S_{SS}, \psi_{SS}, \lambda_{SS})$. The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}^{O}_{SS} = \overset{i=1,\ldots,q}{\underset{Lstvertex^{P},end\_AG}{||}} \mathcal{LS}^{O_i}_{SS} \qquad \mathcal{LS}^{S}_{SS} = \overset{j=1,\ldots,s}{\underset{end\_AG,out\_\lambda,ini\_SS}{\bigsqcup}} \mathcal{LS}^{S_j}_{SS}$$

$$\mathcal{LS}^{OJ}_{SS} = \overset{k=1,\ldots,r}{\underset{end\_AG}{||}} \mathcal{LS}^{OJ_k}_{SS}$$

And the final LGSPN model $\mathcal{LS}_{SS}$ for the subactivity state is now defined by:

$$\mathcal{LS}_{SS} = ((\mathcal{LS}^{OJ}_{SS} \underset{end\_AG}{||} \mathcal{LS}^{S}_{SS}) \underset{end\_AG}{||} \mathcal{LS}^{O}_{SS}) \underset{ini\_SS}{||} \mathcal{LS}^{B}_{SS}$$

3.3 Call states

Call states are a particular case of action states in which its associated entry action is a CallAction, so translation of these elements is quite similar. It must be noted that when a CallAction is executed a set of CallEvents may be generated. For the sake of simplicity, we assume that at most one event is generated, but definition can be extended adding new places in the LGSPN to consider that possibility as well.

Besides, the CallAction may be synchronous or not depending on the value of its attribute *isAsynchronous*, where *synchronous* means that the action will not be completed until the event eventually generated by the action is not consumed by the receiver. In that case, we need a new place and transition in the corresponding LGSPN to model the synchronization (see figure 2, cases 3.a, 3.c and 3.e).

To translate a call state, steps to follow are similar to those described in section 2.2. Given a call state CS,

- If verifies $S.entry.IsAsynchronous = false$ (i.e., its associated CallAction is a synchronous call) we define u as the number of outgoing transitions OS_i of the state (which do not end in a join pseudostate), v the number of outgoing transitions OJS_k that end in a join pseudostate and w the number of self-loop transitions SS_m.
- If verifies $S.entry.IsAsynchronous = true$ -i.e., its associated CallAction is an asynchronous call- we define u' as the number of outgoing transitions OA_j of the state (which do not end in a join pseudostate), v' the number of outgoing transitions OJA_l that end in a join pseudostate, and w' the number of self-loop transitions SA_n.

Also let evx be an event generated by the call action, $evx = S.entry.operation{\rightarrow}ocurrence$. Considering this, we have one LGSPN for each outgoing or self-loop transition t, $\mathcal{LS}^{t}_{CS} = (S^{t}_{CS}, \psi^{t}_{CS}, \lambda^{t}_{CS})$, as shown in figure 2, cases 3.a-3.f. Therefore, we have either $u + v + w$ or $u' + v' + w'$ LGSPN models that need to be combined to get a model of the state CS, $\mathcal{LS}_{CS} = (S_{CS}, \psi_{CS}, \lambda_{CS})$.

The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}_{CS}^{OS} = \underset{Lstvertex^P, Lev^P}{||_{i=1,\ldots,u}} \mathcal{LS}_{CS}^{OS_i} \quad \mathcal{LS}_{CS}^{OA} = \underset{Lstvertex^P, Lev^P}{||_{j=1,\ldots,u'}} \mathcal{LS}_{CS}^{OA_j}$$

$$\mathcal{LS}_{CS}^{OJS} = \underset{ini\_CS, Lev^P}{||_{k=1,\ldots,v}} \mathcal{LS}_{CS}^{OJS_k} \qquad \mathcal{LS}_{CS}^{OJA} = \underset{ini\_CS, Lev^P}{||_{l=1,\ldots,v'}} \mathcal{LS}_{CS}^{OJA_l}$$

$$\mathcal{LS}_{CS}^{SS} = \underset{ini\_CS, Lev^P}{||_{m=1,\ldots,w}} \mathcal{LS}_{CS}^{SS_m} \qquad \mathcal{LS}_{CS}^{SA} = \underset{ini\_CS, Lev^P}{||_{n=1,\ldots,w'}} \mathcal{LS}_{CS}^{SA_n}$$

The final LGSPN for the state $\mathcal{LS}_{CS}$ is defined by one of the two following equations, depending on whether the action was synchronous or not:

$$\mathcal{LS}_{CS} = (\mathcal{LS}_{CS}^{SS} \underset{ini\_CS, Lev^P}{||} \mathcal{LS}_{CS}^{OS}) \underset{ini\_CS, Lev^P}{||} \mathcal{LS}_{CS}^{OJS} \qquad (synchronous)$$

$$\mathcal{LS}_{CS} = (\mathcal{LS}_{CS}^{SA} \underset{ini\_CS, Lev^P}{||} \mathcal{LS}_{CS}^{OA}) \underset{ini\_CS, Lev^P}{||} \mathcal{LS}_{CS}^{OJA} \qquad (asynchronous)$$

3.4 Decisions

Decisions are preprocessed before the AD translation, as it will be mentioned in section 4.1. They are substituted by equivalent outgoing transitions on action states (as shown in figure 3), preserving the properties inherent in performance annotations. Therefore, they do not have to be translated.

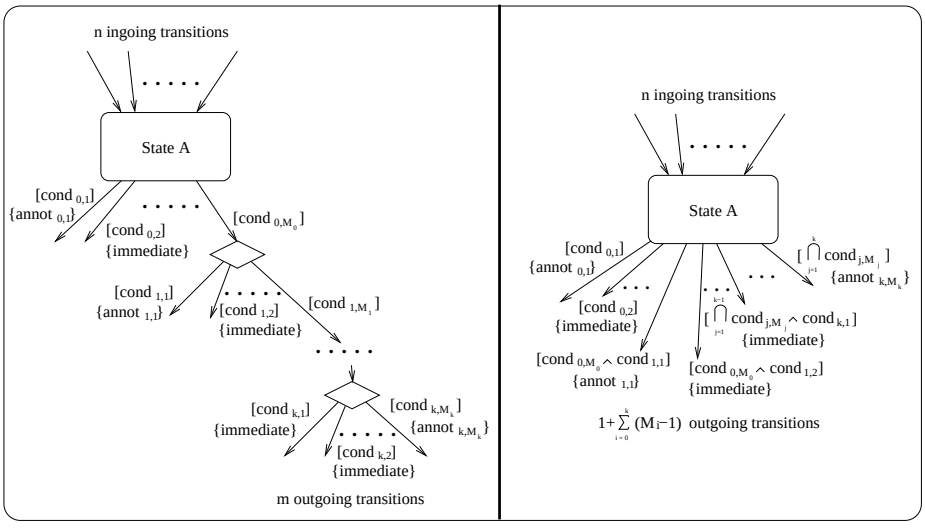


Fig. 3. Decision to LGSPN (Pre-transformation)

3.5 Merges

Merges are used to reunify control flow, separated in divergent branches by decisions (or outgoing transitions of states labelled with guards). Often they are just a notational convention, as reunification may be modelled as ingoing transitions of a state.

Translation of a merge pseudostate M depends on the kind of target element of its outgoing transition. Figure 4 (cases 5.a and 5.b) shows the direct translation of the model, $\mathcal{LS}_M$, according to the condition expressed below.

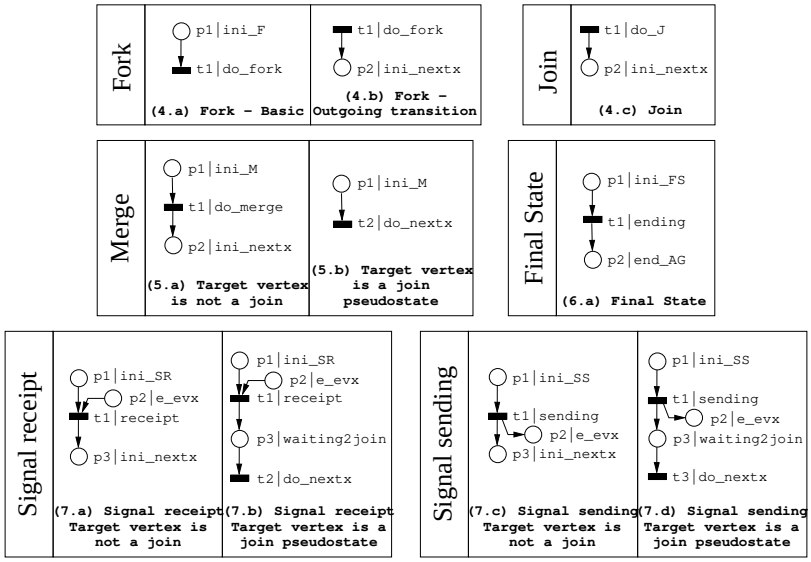


Fig. 4. Fork, Join, Merge, Final State, Signal Sending and Signal Receipt to LGSPN

(a) $\mathcal{LS}_M = \mathcal{LS}'_M \iff (PS.outgoing.target \notin Pseudostate \vee PS.outgoing.target.kind \neq join)$ (to join)
(b) $\mathcal{LS}_M = \mathcal{LS}''_M \iff (PS.outgoing.target \in Pseudostate \wedge PS.outgoing.target.kind = join)$ (not to join)

3.6 Concurrency support items

UML provides two elements to model concurrency in an AD: forks and joins. Their use and meaning do not need further explanation, as they have been commonly explained in classic literature. Translation into LGSPN models is quite simple in both cases.

Given a join pseudostate J, it is translated into the labelled system $\mathcal{LS}_J$, shown in figure 4, case 4.c.

To translate forks, three steps must be followed:

Given a fork pseudostate F let q be the number of its outgoing transitions O_i. Then, according to the translations shown in figure 4, we have a basic LSGPN $\mathcal{LS}_F^B = (S_F^B, \psi_F^B, \lambda_F^B)$ (case 4.a in the figure) and one LGSPN (case 4.b) for each outgoing transition t, $\mathcal{LS}_F^t = (S_F^t, \psi_F^t, \lambda_F^t)$. Therefore, we have $q+1$ LGSPN models that need to be combined to get a model of the pseudostate, $\mathcal{LS}_F = (S_F, \psi_F, \lambda_F)$. The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}_F^O = \underset{do\_fork, Lstvertex^P}{\overset{i=1,\ldots,q}{||}} \mathcal{LS}_F^{O_i}$$

And the final LGSPN $\mathcal{LS}_F$ is composed following the expression:

$$\mathcal{LS}_F = \mathcal{LS}_F^B \underset{do\_fork}{||} \mathcal{LS}_F^O$$

3.7 Initial and final states

Initial pseudostates and final states are elements inherited from UML state machines semantics. However, unlike it happened on UML SMs [6] the initial pseudostate is not translated into a LGSPN model when translating an AD, as no action can be attached to its outgoing transition. On the other hand, final states are translated, but the resulting LGSPN is different from that shown in [6].

Given a final state FS, the LGSPN model $\mathcal{LS}_{FS} = (S_{FS}, \psi_{FS}, \lambda_{FS})$ equivalent to the state is defined according to the translation shown in figure 4, case 6.a.

3.8 Signal sending and signal receipt

Signal sending and signal receipt symbols are control icons. That means they are not really necessary, but are used as a notational convention to specify common modeling matters. In our specific case, these symbols are the only mechanisms we allow to model the processing of external events, and are equivalent to labelling the outgoing transition of a state with a SendAction corresponding to the signal as an effect or with the name of the SignalEvent expected as the trigger event, respectively.

As these symbols are control icons, there is not a metaclass corresponding to this elements in UML metamodel. So we assume that before translating the diagram a unique identificator is assigned to each one of these elements, so when we say $t.target.name$, where t is a incoming transition of the control icon, we are refering to this identificator (instead of the name of the real target StateVertex according to the metamodel).

Given a signal sending/receipt symbol CS, the translation of the symbol depends on whether this target element is a join pseudostate or not:

- If the symbol is a signal sending, then let $SIGS$ be its pre-assigned identificator. Its translation into a LGSPN model $\mathcal{LS}_{SIGS}$ is shown in figure 4, cases 7.c-7.d.

– If the symbol is a signal receipt, then let *SIGR* be its pre-assigned identificator. Its translation into a LGSPN model $\mathcal{LS}_{SIGR}$ is shown in figure 4, cases 7.a-7.b.

It must be noted that, as far as signal sendings is concerned, we have assumed that at most one event is generated for simplicity, but definition can be extended adding new places in the LGSPN to consider that possibility as well.

3.9 Performance-irrelevant constructs

Some elements from ADs are not relevant for performance evaluation, so they are not translated into LGSPN models. These elements are:

– *Swimlanes*, which have no meaning in the dynamics of the system modelled, as they are mechanisms to organize visually the states within the diagram.
– *Action-Object Flow relationships*, as they do not provide any additional concrete information about the behavior of the system.
– *Deferrable events* as, according to our interpretation (see section 2), any event is deferred in an AD (except, obviously, SignalEvents when a signal receipt symbol is found).

4 The System Translation Process

In the previous section we have presented our method to translate every AD element into LGSPN models. Here, we will focus on the whole system translation process, presenting an overview of the steps to follow and allocating the ideas already presented in their own timing. The process includes the complete translation method for ADs and the way to integrate the resulting LGSPN with the ones obtained from the translation of UML SMs and SDs [6].

4.1 Translating activity diagrams into GSPN

As an initial premise we assume that every AD in the system description has exactly one initial state plus, at least, one final state and another state from one of the accepted types (action, subactivity or call state). The translation of an AD can then be divided in three phases, which are presented in the subsequent paragraphs.

Pre-transformations Before translating the AD into a LGSPN model, we need to apply some simplifications to the diagram in order to properly use the translations given in section 3. These simplifications are merely syntactical so the system behaviour is not altered. Most relevant ones are:

– Suppression of decisions. Figure 3 shows a particular case of this kind of transformation. New decisions could be found in any branch of the chaining tree, but the figure has been simplified for the sake of simplicity.

- Suppression of merges / forks / joins chaining, bringing them together into a unique merge / fork / join pseudostate.
- Deducting and making explicit the implicit control flow in action-object flow relationships, where aplicable.
- Avoidance of bad design cases (e.g., when the target of a fork pseudostate is a join pseudostate).

Translation process Once pre-transformations are applied we can proceed to translate the diagram into a LGSPN model. This is done following three steps:

step 1 Translation of each diagram element, as shown in section 2.

step 2 Superposition of the LGSPNs corresponding to the whole set of each kind of diagram elements:

$$\mathcal{LS}_{AG}^{actst} = \overset{AS \in ActionStates}{\underset{Lstvertex^P}{||}} \mathcal{LS}_{AS} \qquad \mathcal{LS}_{AG}^{subst} = \overset{SS \in SubactivityStates}{\underset{Lstvertex^P}{||}} \mathcal{LS}_{SS}$$

$$\mathcal{LS}_{AG}^{calst} = \overset{CS \in CallStates}{\underset{Lstvertex^P, Lev^P}{||}} \mathcal{LS}_{CS} \qquad \mathcal{LS}_{AG}^{merge} = \overset{M \in Merges}{\underset{Lstvertex^P}{||}} \mathcal{LS}_{M}$$

$$\mathcal{LS}_{AG}^{fork} = \overset{F \in Forks}{\underset{Lstvertex^P}{||}} \mathcal{LS}_{F} \qquad \mathcal{LS}_{AG}^{join} = \overset{J \in Joins}{\underset{Lstvertex^P}{||}} \mathcal{LS}_{J}$$

$$\mathcal{LS}_{AG}^{finst} = \overset{FS \in FinalStates}{\underset{end\_AG}{||}} \mathcal{LS}_{FS} \qquad \mathcal{LS}_{AG}^{sigse} = \overset{SIGS \in SignalSendings}{\underset{Lstvertex^P, Lev^P}{||}} \mathcal{LS}_{SIGS}$$

$$\mathcal{LS}_{AG}^{sigre} = \overset{SIGR \in SignalReceipts}{\underset{Lstvertex^P, Lev^P}{||}} \mathcal{LS}_{SIGR}$$

step 3 Working out the LGSPN for the diagram itself by superposition of the LGSPNs obtained in the last step:

$$\begin{aligned}\mathcal{LS}_{AG} = (((((((\mathcal{LS}_{AG}^{sigre} \underset{Lstvertex^P, Lev^P}{||} \mathcal{LS}_{AG}^{sigse}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{finst}) \\ \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{join}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{fork}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{merge}) \\ \underset{Lstvertex^P, Lev^P}{||} \mathcal{LS}_{AG}^{calst}) \underset{Lstvertex^P, end\_AG}{||} \mathcal{LS}_{AG}^{subst}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{actst}\end{aligned}$$

Thanks to this compositional approach, all kind of legal dependencies between diagrams (as looping dependencies) can be processed. E.g., let AG_1 be an activity graph where SS is a subactivity state in it, $SS \in AG_1.transitions.source$, and let AG_2 be the activity graph that the state invokes, $AG_2 = SS.submachine$. Also let SS' be a subactivity state in AG_2, $SS' \in AG_2.transitions.source$, which invokes AG_1, $AG_1 = SS'.submachine$. The superposition operators allows the performance engineer to deal with such syntactical issues.

Post-optimizations Contrasting with pre-transformations, which are mandatory, post-optimizations are optional. Their objective is just to eliminate some

spare places and transitions in the resulting LGSPN so as to make it more attractive without altering its semantics. One example of these kind of transformations would be, in subnets of the LGSPN corresponding to outgoing timed transitions of action states $\mathcal{LS}_{AS}^{OT}$, the removal of the superfluous immediate transitions (and their output place) in case of no conflict.

4.2 Composing the whole system

As it has been stated before, in terms of performance evaluation we use UML ADs exclusively to describe doActivities in SCs or activities inside subactivity states of others ADs. Hence, the merging of nets corresponding to SCs and ADs will be dealt with first.

Let d be the number of ADs used at system description and $Linterfaces^P = \{Lini\_top^P, Lev^P, Lend\_AG^P\}$, where $Lini\_top^P$ is the set of initial places of the LGSPNs corresponding to the ADs and $Lend\_AG^P$ the set of final places of those nets. Now, we can merge the referred LGSPNs by superposition (of places):

$$\mathcal{LS}_{ad} = \mathop{||}_{Linterfaces^P}^{AG \in ActivityDiagrams} \mathcal{LS}_{AG}$$

Now let $\mathcal{LS}''_{sc}$ be the LGSPN corresponding to the translation of the set of SCs in the model. $\mathcal{LS}''_{sc}$ was previously obtained by composition (superposition of places) of the nets obtained for each SC and subsequent removal of sink *acknowledge* places (see [6]).

Then let T_act be the set of transitions in $\mathcal{LS}''_{sc}$ labelled *activity* [6] which represent activities that are described with activity diagrams. $\mathcal{LS}_{sc}$ will be the result of that labelled system with the removal of this set of transitions, $\mathcal{LS}_{sc} = \mathcal{LS}''_{sc} \setminus T\_act$. Ingoing places for these transitions (labelled *end_entry_A* in $\mathcal{LS}''_{sc}$) will be now labelled *ini_top*, where *top* is the name of the first element of the activity diagram AG' that represents the activity, $top = AG'.top.name$. Similarly, outgoing places (labelled *compl_A*) will be now labelled *end_AG'*.

Once done, we can merge the LGSPN systems $\mathcal{LS}_{sc}$ and $\mathcal{LS}_{ad}$:

$$\mathcal{LS}_{sc-ad} = \mathcal{LS}_{ad} \mathop{||}_{Linterfaces^P} \mathcal{LS}_{sc}$$

The resulting net $\mathcal{LS}_{sc-ad}$ often represents the whole system behavior. However, this behavior can be constrained to obtain performance measures for a particular scenario (pattern of interaction). That is done by merging $\mathcal{LS}_{sc-ad}$ and the LGSPN corresponding to a specific SD into a unique LGSPN $\mathcal{LS}$, mainly by synchronization (i.e., superposition of transitions). Paper [6] describes two approaches for doing an analogous operation, using the referred net $\mathcal{LS}_{sc}$ instead of $\mathcal{LS}_{sc-ad}$. Nevertheless, both procedures are still directly applicable to the resulting LGSPN $\mathcal{LS}_{sc-ad}$.

A sample case of the translation of a very simple system is illustrated in figure 5. Two SC and AD models for the system are presented on the left side

of the figure. We have obviated some diagrams of the system description so only part of the resulting LGSPN is included on its right side. That results in the lack of tokens in the initial marking of the net.

The SC represents the life-cycle of an object from the class *car wash machine*, that can be either working or inactive (i.e, waiting for a new car to be washed). The activity performed by the machine when it is working is described by the AD below. As it is shown, the machine works in a different way depending on the amount of money spent by the driver, and can do some tasks simultaneously.

It must be noted that the LGSPN subsystem for the SC has been simplified. In order to proceed to the composition of the LGSPN corresponding to the whole system we should eliminate the transition $t1$ and change the labels of the places $p2$ and $p3$ to *ini_weighcoins* and *end_wash_car*, respectively.

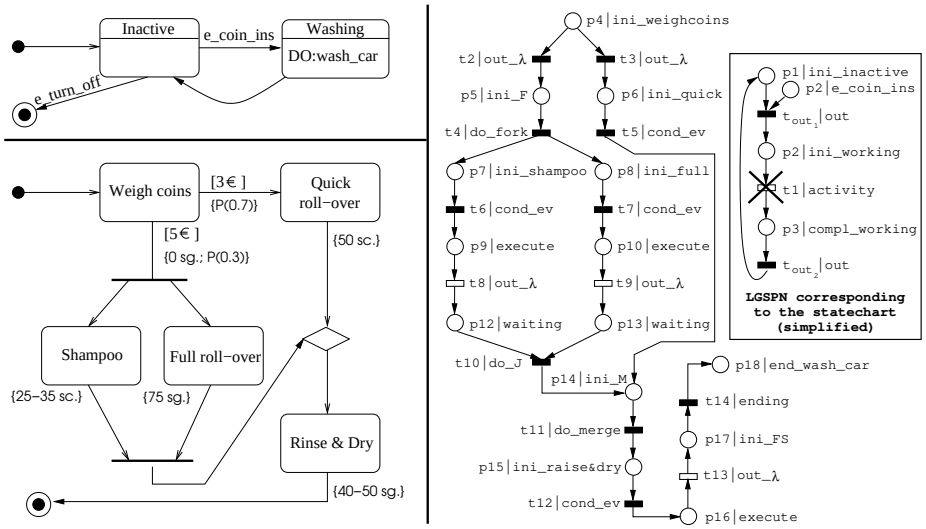


Fig. 5. Car wash machine example

5 Software Performance Tool

To accomplish our objective of successfully integrating techniques of performance evaluation in the software engineering process, an special effort in the automatization of the method is required. To do so, we have developed a CASE tool prototype that generates generalized stochastic Petri nets (GSPN) in a file format [9, 12] directly processable by the GreatSPN tool [12], which is used to make quantitative analysis and obtain performance rates.

The prototype itself provides full capability to model and translate any aspect of ADs as described in this paper, by means of an intuitive, flexible and highly

configurable GUI. Furthermore, support for importing and exporting models in XMI [13] format, standard for CASE tools, is currently in development phase.

In order to easily describe complex systems with a number of diagrams, the tool and the internal file formats are fully project-oriented. This means that every UML element and diagram our tool handles always belongs to a project.

Finally it must be noted that an special effort has been made to obtain highly-legible GreatSPN nets, avoiding superposition of places and transitions. Figure 6 shows a snapshot of a diagram in our tool (the classical 'coffee' example, shown in UML specification [7]) and its resulting translation in GreatSPN, as it was obtained originally.

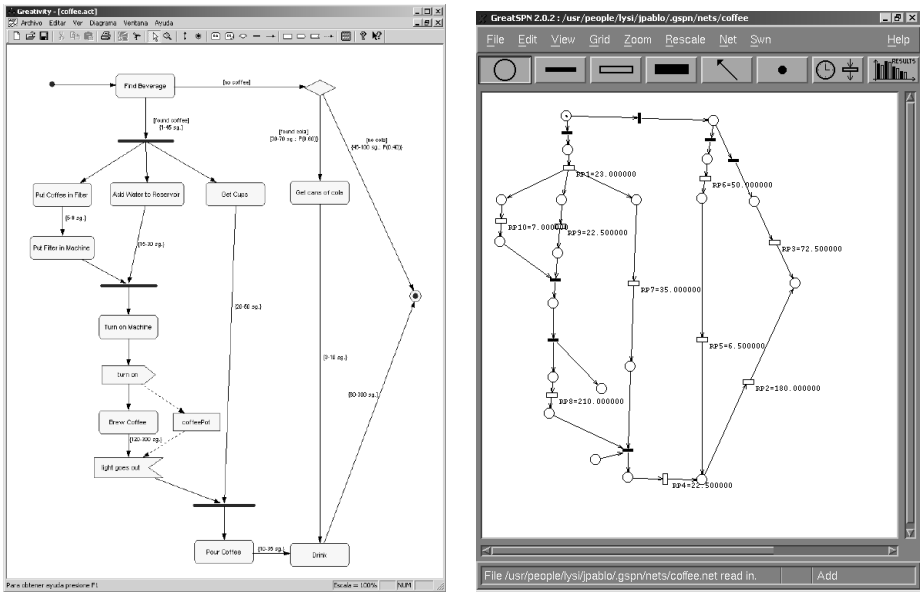


Fig. 6. Tool - Extended coffeepot example and results in GreatSPN

6 Conclusions

The main contributions of this paper can be summarized as follows:

- We have described the kind of annotations suitable to model performance requirements in the context of ADs.
- We have given a translation of the AD (that models a doActivity) into a stochastic Petri net model. In this way, it can be composed with any other stochastic Petri net model that represents a SC that uses the doActivity.
- A formal semantics for the AD is achieved in terms of stochastic Petri nets that allows to check logical properties as well as to compute performance

indices. Obviously, this formal semantics represents an interpretation of the "informally" defined concepts of the UML AD. Our interpretation is focused on the basis that the AD is meant for the description of the doActivities in a SC.
- A prototype tool has been implemented for the Windows® platform. It offers a front end that allows to model all the elements in UML ADs notation in contrast with other tools such as Rose [21] which does not allow to model important features such as signal sending or signal receipt symbols. Performance annotations can be introduced to produce a GSPN system that can be analyzed by the GreatSPN tool [12], therefore it is possible to obtain performance measures in the steady or transient state.

Concerning related work, in [5] can be found a survey of the different approaches for performance evaluation based on UML diagrams. Although there are several works devoted to obtain formal models from the UML SC [16, 15, 22] or the UML SD [24, 8, 3], some of them with performance evaluation purposes, the AD has not been studied yet so intensely. But an interesting work can be found in [11], where a formal semantics for the AD is given based on the STATEMATE semantics [14] of the statecharts.

To date it is not possible to compare our tool because to our knowledge there exist three tools [4, 2, 18] for performance evaluation based on UML but they do not use stochastic Petri nets as performance models. On the other hand, although DSPNExpress2000 [17] claims to be, it seems that only very simple SCs can be modelled with this tool. In SimML [4], simulation queuing networks models [20] for performance evaluation are obtained from UML class diagram and SD, while in the PERMABASE project [2] models for simulation are obtained from UML SD and class and deployment diagrams.

As future work we are working on the following open issues:

- With respect to UML ADs, conditional forks and more complex external event processing support, especially important to resolve the problem of 'uninterruptable' activities due to the use of action states.
- Extension of the prototype tool to support SCs and SDs in order to increase the expressivity at system description.
- Possibility of processing XMI files in our CASE tool prototype, in order to import models from other CASE tools and thus ensure compliance with current standards.

A Formal definition of composition of GSPNs

Place and transition superposition of two ordinary labeled GSPNs. Given two LGSPN ordinary systems $\mathcal{LS}_1 = (S_1, \psi_1, \lambda_1)$ and $\mathcal{LS}_2 = (S_2, \psi_2, \lambda_2)$, the LGSPN ordinary system $\mathcal{LS} = (S, \psi, \lambda)$:

$$\mathcal{LS} = \mathcal{LS}_1 \underset{L_T, L_P}{||} \mathcal{LS}_2$$

resulting from the composition over the sets of (no τ) labels L_T and L_P is defined as follows. Let $E_T = L_T \cap \lambda_1(T_1) \cap \lambda_2(T_2)$ and $E_P = L_P \cap \psi_1(P_1) \cap \psi_2(P_2)$ be the subsets of L_T and of L_P, respectively, comprising place and transition labels that are common to the two LGSPNs, P_1^l (T_1^l) be the set of places (transitions) of $\mathcal{LS}_1$ that are labeled l and $P_1^{E_P}$ ($T_1^{E_T}$) be the set of all places (transitions) in $\mathcal{LS}_1$ that are labeled with a label in E_P (E_T). Same definitions apply to $\mathcal{LS}_2$.

Then: $T = T_1 \backslash T_1^{E_T} \cup T_2 \backslash T_2^{E_T} \cup \bigcup_{l \in E_T} \{T_1^l \times T_2^l\}$, $P = P_1 \backslash P_1^{E_P} \cup P_2 \backslash P_2^{E_P} \cup \bigcup_{l \in E_P} \{P_1^l \times P_2^l\}$, the functions $F \in \{I(), O(), H()\}$ are equal to:

$$F(t) = \begin{cases} F_1(t) & \text{if } t \in T_1 \backslash T_1^{E_T} \\ F_2(t) & \text{if } t \in T_2 \backslash T_2^{E_T} \\ F_1(t_1) \cup F_2(t_2) & \text{if } t \equiv (t_1, t_2) \in T_1^{E_T} \times T_2^{E_T} \wedge \lambda_1(t_1) = \lambda_2(t_2) \end{cases}$$

where $\cup$ on the third line is the union over sets. Functions $F \in \{\Pi(), W()\}$ are equal to:

$$F(t) = \begin{cases} F_1(t) & \text{if } t \in T_1 \backslash T_1^{E_T} \\ F_2(t) & \text{if } t \in T_2 \backslash T_2^{E_T} \\ min(F_1(t_1), F_2(t_2)) & \text{if } t \equiv (t_1, t_2) \in T_1^{E_T} \times T_2^{E_T} \wedge \lambda_1(t_1) = \lambda_2(t_2) \end{cases}$$

The initial marking function is equal to:

$$M^0(p) = \begin{cases} M_1^0(p) & \text{if } \ p \in P_1 \backslash P_1^{E_P} \\ M_2^0(p) & \text{if } \ p \in P_2 \backslash P_2^{E_P} \\ M_1^0(p_1) + M_2^0(p_2) & \text{if } \ p \equiv (p_1, p_2) \in P_1^{E_P} \times P_2^{E_P} \wedge \psi_1(p_1) = \psi_2(p_2) \end{cases}$$

Finally, the labeling functions for places and transitions are respectively equal to:

$$\psi(x) = \begin{cases} \psi_1(x) & \text{if } \ x \in P_1 \backslash P_1^{E_P} \\ \psi_2(x) & \text{if } \ x \in P_2 \backslash P_2^{E_P} \\ \psi_1(p_1) & \text{if } \ x \equiv (p_1, p_2) \in P_1^{E_P} \times P_2^{E_P} \wedge \psi_1(p_1) = \psi_2(p_2) \end{cases}$$

$$\lambda(x) = \begin{cases} \lambda_1(x) & \text{if } \ x \in T_1 \backslash T_1^{E_T} \\ \lambda_2(x) & \text{if } \ x \in T_2 \backslash T_2^{E_T} \\ \lambda_1(t_1) & \text{if } \ x \equiv (t_1, t_2) \in T_1^{E_T} \times T_2^{E_T} \wedge \ \ \lambda_1(t_1) = \lambda_2(t_2). \end{cases}$$

Place and transition superposition and simplification of two ordinary labeled GSPNs. Given two LGSPN ordinary systems $\mathcal{LS}_1 = (S_1, \psi_1, \lambda_1)$ and $\mathcal{LS}_2 = (S_2, \psi_2, \lambda_2)$, the LGSPN ordinary system $\mathcal{LS} = (S, \psi, \lambda)$:

$$\mathcal{LS} = \mathcal{LS}_1 \bigsqcup_{L_T, L_P} \mathcal{LS}_2$$

resulting from the composition over the sets of (no τ) labels L_T and L_P is defined as follows. Let $E_T = L_T \cap \lambda_1(T_1) \cap \lambda_2(T_2)$ and $E_P = L_P \cap \psi_1(P_1) \cap \psi_2(P_2)$ be the subsets of L_T and of L_P, respectively, comprising place and transition labels that are common to the two LGSPNs, P_1^l (T_1^l) be the set of places (transitions)

of $\mathcal{LS}_1$ that are labeled l and $P_1^{E_P}$ ($T_1^{E_T}$) be the set of all places (transitions) in $\mathcal{LS}_1$ that are labeled with a label in E_P (E_T). Same definitions apply to $\mathcal{LS}_2$.

Then: $T = T_1 \backslash T_1^{E_T} \cup T_2 \backslash T_2^{E_T} \cup \bigcup_{l \in E_T} \{T_1^l \times T_2^l\}$, $P = P_1 \backslash P_1^{E_P} \cup P_2 \backslash P_2^{E_P} \cup \bigcup_{l \in E_P} \{P_1^l \times P_2^l\}$, the functions $F \in \{I(), O(), H(), \Pi(t), M^0(), \psi(), \lambda()\}$ are defined exactly as it was made for the last operator, whereas function W(t) is equal to:

$$W(t) = \begin{cases} W_1(t) & \text{if } t \in T_1 \backslash T_1^{E_T} \\ W_2(t) & \text{if } t \in T_2 \backslash T_2^{E_T} \\ W_1(t_1) + W_2(t_2) & \text{if } t \equiv (t_1, t_2) \in T_1^{E_T} \times T_2^{E_T} \wedge \lambda_1(t_1) = \lambda_2(t_2) \end{cases}$$

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley Series in Parallel Computing - Chichester, 1995.
2. D. Akehurst, G. Waters, P. Utton, and G. Martin. Predictive Performance Analysis for Distributed Systems - PERMABASE position. In *One Day Workshop on Software Performance Prediction extracted from Designs*, Heriot-Watt University, Edinburgh, November 1999.
3. F. Andolfi, F. Aquilani, S. Balsamo, and P. Inverardi. Deriving performance models of software architectures from message sequence charts. In *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, pages 47–57, Ottawa, Canada, September 2000. ACM.
4. L.B. Arief and N.A. Speirs. A UML tool for an automatic generation of simulation programs. In *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, pages 71–76, Ottawa, Canada, September 2000. ACM.
5. S. Balsamo and M. Simeoni. On transforming UML models into performance models. In *Proceedings of the Workshop on Transformations in UML, ETAPS 2001*, April 7th 2001.
6. S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In *Third International Workshop on Software and Performance (WOSP2002)*, Rome, Italy, July 2002. ACM. To appear.
7. G. Booch, I. Jacobson, and J. Rumbaugh. OMG Unified Modeling Language specification, September 2001. version 1.4.
8. J. Cardoso and C. Sibertin-Blanc. Ordering actions in sequence diagrams of UML. In *Proc. of* 23^{th} *Int. Conf. on Information Technology Interfaces - ITI2001*, Pula, Croatia, 2001.
9. G. Chiola. GreatSPN 1.5 software architecture. Technical report, Università di Torino, April 1991.
10. S. Donatelli and G. Franceschinis. PSR Methodology: integrating hardware and software models. *LNCS 1091, in Proceedings Appl. and Theory of PNs, Osaka, Japan*, pages 133–152, June 1996.
11. R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In Heinrich Hußmann, editor, *Fundamental Approaches to Software Engineering (FASE2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2001.
12. The GreatSPN tool. `http://www.di.unito.it/~greatspn`.

13. Object Management Group. XML Metadata Interchange (XMI) specification, January 2002. version 1.2.
14. D. Harel and A. Naamad. The STATEMATE semantics of the statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
15. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*, pages 331–347. Kluwer, 1999.
16. J. Lilius and I.P. Paltor. The semantics of UML state machines. *Technical report no.273 - Turku Centre for Computer Science, Finland*, May 1999.
17. C. Lindemann, A. Thummler, A. Klemm, M. Lohmann, and O.P. Waldhorst. Quantitative system evaluation with DSPNexpress 2000. In *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, pages 12–17, Ottawa, Canada, September 2000. ACM.
18. J. Medina, M. González, and J. M. Drake. MAST-UML: Visual modeling and analysis suite for real-time applications with UML. `http://mast.unican.es/umlmast/`.
19. J. Merseguer, J. Campos, and E. Mena. Analysing internet software retrieval systems: Modeling and performance comparison. *Wireless Networks: The Journal of Mobile Communication, Computation and Information*, 2002. To appear.
20. M.K. Molloy. *Fundamentals of Performance Modelling*. Macmillan, 1989.
21. Rational Software Corporation, 2002. http://www.rational.com.
22. A.J.H. Simons. On the compositional properties of UML statechart diagrams. In *Proceedings of the Rigorous Object-Oriented Methods,ROOM 2000*, January 2000.
23. C. U. Smith. *Performance Engineering of Software Systems*. The Sei Series in Software Engineering. Addison–Wesley, 1990.
24. A. Tsiolakis. Integrating model information in UML sequence diagrams. In *Proceedings GT-VMT2001, Electronic Notes in Theoretical Computer Science*, July 2001.
25. M. Woodside, C. Hrischuck, B. Selic, and S. Bayarov. A wide band approach to integrating performance prediction into a software design environment. In *Proceedings of the 1st International Workshop on Software Performance (WOSP'98)*, pages 31–41, 1998.

Design and implementation of High Availability Routing for Linux: HARL*.

Luis Irún-Briz, José M. Bernabéu-Aubán, and Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática
Polytechnic University of Valencia.
Camino de Vera s/n. 46022. Valencia. Spain.
E-mail: {lirun,josep,fmunyoz}@iti.upv.es

Abstract Clustering techniques are commonly used to provide services to the internet. In a cluster, as occurs in many other systems, a single node (known as router or even firewall) is used to interconnect the internet to the system. Some proposals for "improved-availability routers" drops existing connections when a failure in the router occurs. In contrast, our HARL (High Available Router for Linux) router consist of a stateful router/firewall masquerading system guarantees that connections do not close as a result of a failure. In addition, HARL is capable of recovering from partial failures in a bounded time, contributing to the "high availability" of the service provided by servers behind the firewall. This paper presents in detail the design of HARL, and its Linux-based implementation, with its improved IP-masquerading algorithms, and its recovery protocol.

1 Introduction

Many of the services available now a days through the internet, as *Server Farms* [11], need to operate on a continuous base. Such services, thus, incorporate some sort of high availability approach, typically using a cluster to replicate data and/or activity.

The services provided by such clusters must be made available to the *external world* in a consistent way, being, thus, desirable that the composition of the cluster be *hidden* to its clients [5]. In this way, the client views the system as a single *host* providing a set of services.

A common approach to hide a cluster of machines consists in interposing an IP router performing[8] what is commonly known as Network Address Translation (NAT) [6] [10].

In the Internet model, IP datagram forwarders are commonly used to interconnect several network. These *IP datagram forwarders* are called [13] routers or IP routers. In this document, every use of the term router is equivalent to IP router. The term gateway is commonly used in many other Internet documents referring to routers.

* This work has been partially supported by CICYT grants TIC99-0280-C02-01 and TIC2000-1246

Unfortunately, in many cases the router itself becomes a single point of failure in these systems, which makes it necessary to introduce some mechanism to provide the router itself with some degree of availability, at the least guaranteeing continuous connectivity with the cluster.

In order to provide a highly available router some degree of replication must be introduced. Current proposals (i.e. see [11]) are capable of recovering in the presence of a failure, but on such event, ongoing connections are closed, as no state is shared among the replicas.

In many applications (commercial and otherwise), it is important to keep alive established connections. In some cases, suddenly closing the connection leads to loss of data, and the escalation of the problem to the user accessing the services. In others, when applications are specially coded for that, an unexpected connection close can lead to appreciable degradation in performance. High available routers capable of maintaining connections open let clusters provide a more efficient and reliable service. In addition, the client applications can be severely simplified, as it becomes less critical to keep track of eventual connection failures.

Some commercial companies have commercialized fault-tolerant routers (i.e. CISCO, Ltd.) using special hardware to achieve the replication of the router components. This explains the relatively high purchasing cost of those systems, and their low rate of introduction in the market.

The approach we have taken with HARL (High Available Router for Linux) addresses the problem of providing a highly available router, by using existing, very accessible technology (Linux), and implementing on top of it the protocols necessary to provide the needed guarantees, with a minimum of hardware set-up. The guarantees provided by HARL are:

- Persistence of the existing connections over a failure.
- Bounded full-recovery time from a failure in the router system.
- Null overhead for the internal or external networks.
- Scalability in the redundancy of the router system.

This paper presents the design of HARL as well as the algorithms and protocols it uses.

The rest of this paper is structured as follows. Section 2 introduces the basics of the routing techniques, and presents the architecture of our HARL router. In section 3 the HARL implementation is described, and section 4 describes other details as Guaranties offered by HARL and the configuration mechanisms. Finally, we present some concluding remarks in section 5.

2 HARL Overview

An IP router is a physical device that performs the network layer forwarding function of the Internet protocol suite. These devices have been commonly known as *routers* or *gateways*. The main difference between a router and other switching device is [8] that a router examines the IP header of each received IP packet,

and performs the adequate translation in the packet headers before passing these packets to the other network segment.

An IP router does not act just as a mere redirector of IP packages, but it also performs transformations of the IP packages during the redirection[6]. These techniques are known as Network Address Translation (NAT).

IP routers have been widely used during the last decade to provide needed functionality to the organizations that wish to be connected to the Internet.

The use of a router has three main advantages:

- It allows many computers to access to the Internet through a unique IP address.
- A router can be used to centralize the security management of a subnetwork.
- A router can simplify the implementation of advanced techniques for load balancing (centralizing the decision point)

As the routing process needs to keep state, a router becomes a critical, single point of failure, which is not desirable. HARL circumvents this problem replicating the router state using the primary backup approach, giving the system a more robust structure.

2.1 NAT: IP Masquerading

Some time ago, the Internet community centered its attention on routing techniques in order to improve the existing limitations and restrictions [6] and [13]. These techniques are commonly known as NAT (Network Address Translation), and its use and implications are discussed in [10]. This lead to the adoption by the routers of NAT techniques.

NAT routers inspect the IP addresses contained in incoming packets, and map them from one *address administration* to another one. Due to the location of this information within the packets (at the IP headers), the NAT engine is unaware of the applications the packets belong to.

Generic NAT routers makes accessible from the Internet a limited number of IP addresses, while keeping hidden the IP addresses of a much larger number of internal machines connected *behind* the router. Typically, the router acts as an exit gateway to the internet for all machines it hides.

In general, when a router running NAT receives an outgoing packet from an internal machine "M", it takes an IP address from a pool, and changes the IP-header before rewriting the packet in the external segment. During a period of time, incoming packets received from the external segment to this IP address will be routed to the internal machine "M". In order to allow external machines to initiate connection to hosts at the internal network, NAT rules must be defined to control the translation of IP addresses for incoming packets.

When NAT techniques are applied to systems where a unique IP address is published, this is known as IP-Masquerading. Figure 1 shows an example of IP-Masquerading rewriting. When the router receives a packet from an *external* machine to subnet "S", it changes the IP headers in such a way that the source IP

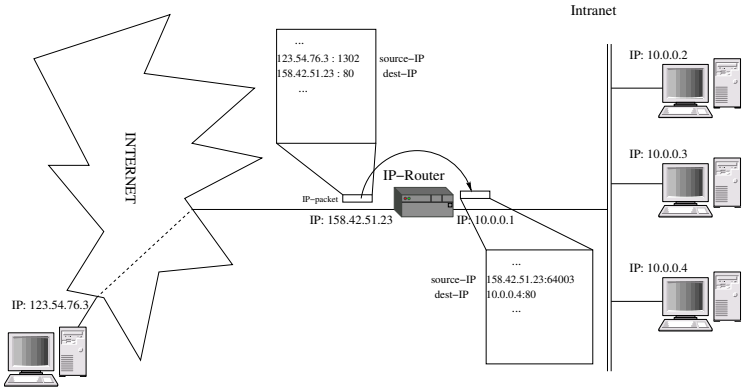


Figure1. Router using NAT

address shown in the modified packet points to the router, and the destination IP address points to the internal machine. The ports in the IP packet are rewritten in an analogous way.

The translated incoming packets are eventually replied to the external machine. In order to perform a sensible translation of the outgoing message (i.e. one that guarantees that the packet is sent to the intended receiver), the router needs to *remember* the original source of the packet. It is thus necessary for the router to keep the state of ongoing packet exchanges to properly reconstruct outgoing reply packets.

The use of NAT techniques introduces several troubles[13] concerning to certain specific protocols (as FTP). In these protocols, the data transmitted by the applications contains information related to the IP protocol. Common NAT techniques only modify the IP header of the packages, not managing the application data (wich may remain outdated). In order to correct these protocol-specific disadjustments, it is needed to perform a per-protocol filtering of these communication.

2.2 Common Redundant Router vs. HARL redundancy

As shown above, the NAT approach requires the router to maintain state. Furthermore, the router becomes a critical point of failure in the system: when the router fails, the entire system becomes unavailable.

In general, to reduce the danger of connectivity loss it is necessary to use redundant routers, i.e., routers consisting of various routing components, each one with assumed independent failure modes (see figure 2). At each point in time, only one router component, the *primary*, is active, presenting the IP address of the router to the network. The other router components (the *backups*) just wait until the active one fails. When the primary fails, one of the backups is *promoted* to primary and is activated. To activate the new primary, it is necessary for it to steal the IP address of the router, previously used by the now failed old primary.

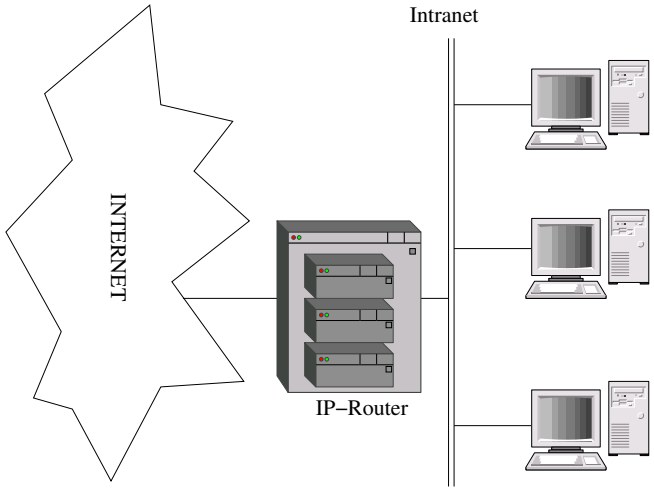


Figure2. Subnet using replicated router

In order to carry out the above approach two problems must be solved: primary failure detection, and IP address takeover by the new primary.

Common redundant routers typically present just one backup. Thus, monitoring proceeds by pinging, and failure detection is implemented by the use of time-outs.

Additionally, IP *takeover* is performed by the new primary by doing an *ARP spoofing* operation. ARP packets are sent to help map between IP addresses and Ethernet MAC addresses. ARP spoofing consists on the suplantation of the identity (the MAC address) of an ethernet destination performing a periodical sent of special ARP packets (known as gratuitous ARP), notifying everyone in each segment of the new MAC address associated with the router's IP address.

This approach for redundant routers permits the new primary to resume the activities of the router after a failure occurs, directing new packets correctly. However, this simple approach has the following problems:

- It is not easy to increase the number of backups for increased availability.
- As no state is shared between primary and backup, after a failure, the new primary does not have information on the existing connections, causing their loss.
- ARP spoofing is not a desirable technique, because it consumes bandwidth of each segment the router is connected to.

HARL's design sets out to approach the above problems by a more rigorous approach to fault detection and handling, and a different approach to IP stealing.

Dealing with failures in distributed systems is plagued with inherent difficulties [2]. In some cases (asynchronous networks) those difficulties make it impossible to solve certain problems [3]. However, under our setting of a closely

controlled redundant router we can relax the asynchronous model, making it possible to implement failure detectors [1].

Failure detection is carried out in HARL by using a simple cluster membership monitor protocol (CMM) [7]. This approach allows the router configuration to scale arbitrarily.

Additionally in HARL, primary and backups cooperate to consistently maintain replicated routing tables using a basic primary-backup approach [4] [9]. When the primary fails, the CMM guarantees that the promoted backup will have a consistent copy of the routing tables. The new primary can then continue routing where the old primary stopped, thus keeping existing connections alive.

Finally, HARL avoids the use of ARP spoofing by the expeditive and simple means of letting the promoted backup modify its MAC address. In this way, the whole redundant router always presents the same MAC and IP address to the rest of the world, making it undistinguishable without consuming any network bandwidth.

2.3 HARL Architecture

HARL's approach consists on replicating a basic router, using a primary backup approach to maintain the state for the routing process. This allows the router system to preserve its state in the presence of failures.

Router access to the networks connected to it (both internal and external) is performed by means of virtual interfaces (with their own MAC address). This approach avoids the bandwidth consumption, and speeds up the failure-recovering process, because no additional message is needed for the backup to promote to *primary*.

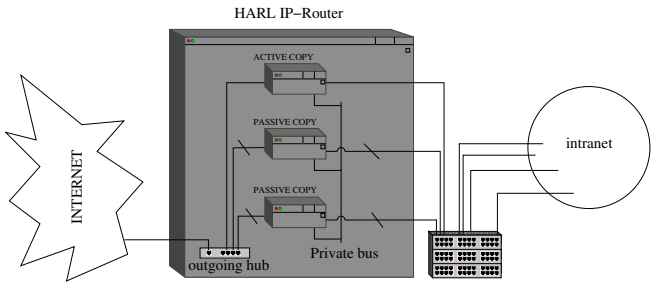


Figure3. HARL router architecture

Figure 3 shows the basic layout of a HARL router.

The failure model for the nodes in the system has been assumed to be *stop*. In addition, a node failure implies a complete disconnection of the node from the system.

Hardware configuration In HARL, every component replica of the router has three network interfaces. The *external* interface links the replica to the external network (the Internet). The *internal* interface provides connectivity to the internal segment (the intranet of the organization). The third interface, is a *private* interface, connected to a private network used only by the replicas. HARL replicas use the private network to run HARL's protocols, thus avoiding consuming bandwidth from the "production" networks.

At any given point in time, only the primary has its external and internal interfaces active. The backups do not use their internal and external interfaces. The private interface of every replica in the HARL system is active running HARL's group membership protocol, MMM on top of the communication protocol, MGP, which, in turn, runs on top of the physical media.

The non-private interfaces of all replicas are attached to an extremely simple HUB. This HUB becomes a single point of failure in the HARL system, but the behaviour of the replicas allows the implementation of this HUB using a simple (and thus extremely unlikely to fail) physical union of the lines of the Ethernet cable. This can be done, as in HARL only the primary activates its non-private interfaces, and the protocols guarantee that there is only one primary at any given time, so a collision in this subsegmet becomes virtually impossible.

The private segment of the HARL system can be implemented using a variety of physical media. In our prototype we have used a simple Ethernet 10Base2 interface (a simple coaxial segment bus), but other alternatives such *firewire* and *usb* are also possible. Every replica in the system can communicate with all the others using this segment. The segment acts as a *warm bus*, that is, it is possible to attach or detach a number of hosts to the bus without the need of restarting any component in the system. HARL's protocols are bandwidth-lean, thus the bandwidth requirements on the private network are non-critical.

It is possible to eliminate from the system the private segment, at the cost of introducing an overhead on the internal network. In this case, the private interfaces cannot be joined using the *extremely simple HUB*, because the HARL protocols would always be running in the private segment for every replica, and a more complex HUB (or a 10Base2 bus) becomes necessary.

Software configuration Our HARL system has a set of software pieces included to provide specific functionalities. These components are:

- Communication layer: MGP
- Network interfaces configuration: MAC addressing and IP configuration
- "Primary copy" IP-masquerade algorithm
- Detection of failures of the primary node: MMM
- Leader promotion on failures

MGP (Minimal Group Protocol) is the protocol used by the HARL components to inter operate. In section 3.2 we show this protocol in detail. The MGP protocol is an Ethernet-based protocol, so only MAC addresses are needed for an MGP packet to be sent, and received by a set of destination nodes.

The *Network interfaces configuration* consists of a set of routines allowing the system to override the MAC address of an Ethernet card, and force the kernel to use another MAC as the native MAC of Ethernet interface corresponding to this card (Section 4.1).

The *"Primary copy" IP-masquerade algorithm* lets the primary notify every backup of the changes in its *routing tables*.

Every backup should be able to learn of the failure of the primary. The *MMM (Membership Maintenance Manager)* protocol described in section 3.3 is used for that.

When the primary fails, the system reconfigures itself after a bounded period of time using the leader promotion protocol, described in section 3.4.

3 HARL Implementation: HA-IPMASQ Module

HARL has been implemented by a loadable kernel module. This allows the system to simplify the configuration process, and makes HARL more flexible and independent of the kernel version. However, a minimal kernel aid has been necessary to couple HARL's routines within the module with the kernel IP masquerading process.

3.1 Core Kernel Support: the IP-Masq Modification

The modifications performed in the kernel are centered around the IP-Masquerade subsystem. This is contained in the file `ip_masq.c`. In its common behavior, when a new connection reaches the router, a routine is fired in this subsystem. This routine manages the modification of a set of routing tables (containing the routing information, timeouts for its invalidation, etc. . .) that are used for rerouting the packets corresponding to each connection. HARL's module modifies this subsystem, providing a *hook* mechanism, by which the masquerade subsystem invokes a set of routines when the routing tables are modified.

A list is used to allow modules to insert their processing routines into the routing algorithm. The "process list" is implemented as follows:

```
struct t_ip_process {
        struct t_ip_process *next;
        int (*ip_masq_hash)( struct ip_masq *ms, int calc_hash);
        int (*ip_masq_unhash)( struct ip_masq *ms, int calc_unhash);
        unsigned long data;
} t_ip_process;
struct t_ip_process *ip_process=NULL;
```

The standard routines that modify the routing tables (`ip_masq_hash` and `ip_masq_unhash`) have been modified in order to notify the changes to the installed callbacks. The original routines are published as kernel symbols, in order to simplify the access to the tables during the synchronization process. The renamed routines have been published as `just_ip_masq_hash` and `just_ip_masq_unhash`.

```
static int ip_masq_hash(struct ip_masq *ms) {
        int i=just_ip_masq_hash(ms);
        if (ip_process!=NULL && ip_process -> ip_masq_hash!=NULL)
            i=ip_process - > ip_masq_hash(ms, i);
        return i;
}

static int ip_masq_unhash(struct ip_masq *ms) {
        int i=just_ip_masq_unhash(ms);
        if (ip_process!=NULL && ip_process -> ip_masq_unhash!=NULL)
            i=ip_process - > ip_masq_unhash(ms, i);
        return i;
}
```

The callback handling routine invokes the rest of the callbacks in the list (using the field `next` of the `ip_process` variable) in order to chain the registered managers. In our case, a unique manager is registered.

Additionally, another symbol is published to enable its use from the modules: `__ip_masq_lock`. This is the lock used to control the access to the routing tables. When a routine needs to modify these tables, it is necessary to perform a lock over `__ip_masq_lock` before the manipulation, and when this manipulation is ended, the routine should unlock the `__ip_masq_lock` variable.

The HARL-specific routines implemented to manage the table replication are attached to the main system using the `ip_process` variable. Our implementation consists on sending a broadcast message to all backups. This message contains the information of the change made in the routing tables (it can be an insertion or a deletion of a row in the tables). The broadcast is performed using the MGP protocol.

3.2 MGP Protocol

HARL uses a private transport layer as its main communication mechanism for the replicas in the system (a transport layer consists of the physical medium the nodes use to transmit its messages). In general, this layer should be implemented as a bus, with the ability of warm reconfigurations (the bus should be able to suffer an arbitrary number of attachment or detachment of nodes, without producing interruptions in the communication of the rest of nodes). HARL uses a specific Ethernet protocol for this kind of connection bus. In order to simplify the HARL design, the used transport protocol should satisfy at least the following characteristics:

- Provide unique node identifiers to the replicas
- Implement a connectionless protocol
- Be based on fixed-sized messages
- Two send primitives: point-to-point and atomic broadcast
- One receive primitive

The MGP protocol (or Minimal Group Protocol) have been implemented to be used as transport layer. It whould be possible to use other existing protocols as IGMP (or even UDP), but MGP has been designed in order to implement the minimal functionality, minimizing the introduced overhead.

The *node identification* is provided by the Ethernet layer (using the MAC address of the private Ethernet interface of each node), thus providing a total order among node identifiers (i.e. it is possible to say, for an arbitrary couple of nodes: $id(Node_i) > id(Node_j)$, or $id(Node_i) < id(Node_j)$.

The protocol doesn't need to establish any connection, as the information flow will always run from the primary to the backups.

The messages transmitted between the nodes have a well-known size, thus the network load is characterized by the number of exchanged messages.

The `send_to` primitive allows a node to send a message to another node, given its MAC address. The MAC address of the origin node is included into every message. The incoming message is enqueued into a buffer in the message receiver. The *receive* primitive dequeues a message from the incoming message buffer.

A `broadcast` primitive is provided by MGP to permit a node to send a message atomically to the rest of the nodes. By atomically, we mean that either the message is delivered to every replica or to none. In addition, the `broadcast` operation guarantees total message ordering, that is, if two messages m_1 and m_2 are broadcast by two nodes ($t_{send}(m_1) < t_{send}(m_2)$), these messages are delivered by **every** destination node exactly in the same order ($t_{del}(m_1) < t_{del}(m_2)$).

It is important to see that the guarantees provided by MGP can be achieved using a unique Ethernet segment. In this kind of layer [12], only one sender can successfully use the bus at a time, and every node can be listening from the bus any successfully sent message. So a single message can be sent to every node in the segment at the same cost (for the sender) of a point-to-point message.

In HARL, all nodes have an identical computational capacity, and their communication interfaces are also identical.

The Ethernet messages incoming to a node are queued into a buffer. If the receiving nodes can process their incoming messages faster than the sender node emits them, the Ethernet bus ensures that, in absence of physical failures, no messages are lost.

It can be proved that judicious configuration of buffer sizes on HARL make it impossible for a node to lose the messages employed in HARL's protocols.

3.3 MMM: The Membership Maintenance Manager

HARL uses an extremely simple algorithm to achieve the director failure detection, and the director election. This algorithm (MMM) makes use of MGP in order to perform the steps in which it is involved. The algorithm does not care for failures of backups, only failures of the primary are of interest.

In the MMM distributed algorithm, we have a set of nodes $N_1 \cdots N_n$. These nodes modify their state during the system execution. The states of a node within MMM are:

- *PROMOTION*, reached whenever a reconfiguration is detected.
- *JUSTSLAVE*, the node knows about an existing master, or about a better node for mastering, and then is about to configure itself as *SLAVE*.
- *SLAVE*, the node knows about an existing master, or about a better node for mastering, and it will listen *MASTER* messages.
- *MASTER*, the node has passed the *PROMOTION* state, so it is *MASTER*.
- *UNKNOWN*, the state of the node is temporally unknown. This is the initial state.

In order to communicate its state, the nodes make use of a set of MGP messages:

- *MASTERING*, sent whenever a node is entering the group
- *PING*, common heartbeat broadcast message
- *SLAVING*, sent to the master asking for the cache
- *NEW*, sent to new slave nodes in response to a *SLAVING* message
- *UPDATE*, broadcast to every slave node.

The MMM algorithm can be shown in figure 4.

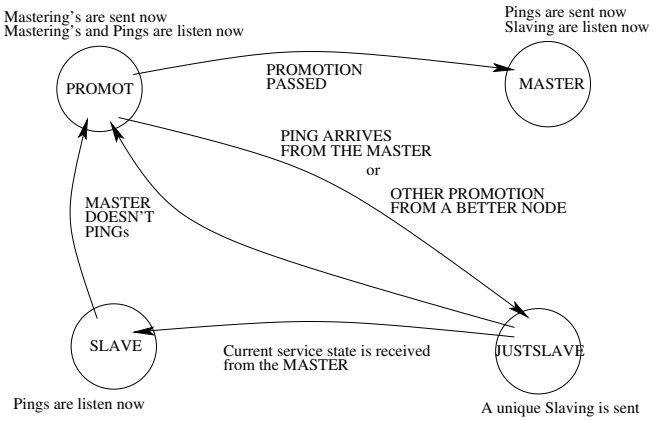


Figure4. MMM state diagram

Every message sent by the nodes includes in its header a number identifying the incarnation of the system in which the message was sent. This incarnation number is increased after every reconfiguration [1] in the system.

When a node N_i is attached to the system, its first state is *PROMOTION*. This state has a well-known duration, and is used to ensure that only one of the nodes can promote to the *MASTER* state.

[1] In the rest of the section, we use the term *reconfiguration* to designate a change of primary assignment.

During this state, the node N_i broadcasts periodically messages of the type *MASTERING*. If N_i receives a *PING* message from another node N_j (the current MASTER),N_i transits its state to *JUSTSLAVE*, because it knows about another node (N_j) that is the current master. If the received message from N_j is a *MASTERING* message, node N_i will compare its identifier ("i") and the identifier of the other node ("j"). The node N_i will only continue if $i < j$. In other case, the node N_i transits to the *JUSTSLAVE* state.

When a node reaches the *MASTER* state, it starts the periodical broadcast of *PING* messages. In addition to these heart-beating task, the master services the *SLAVING* messages sent by new slaves in the system. When a *SLAVING* message is received from N_i, the master sends the current state of the routing tables to N_i, in order to synchronize the data state of the new slave. Whenever the primary node (the unique node in *MASTER* state) changes its state, it broadcasts an *UPDATE* message to the slaves, thus making a backup of its state within the backup replicas.

If a node N_i falls into the *JUSTSLAVE* state, it waits until a PING message is received in order to identify the master node. Then, the node N_i sends a *SLAVING* message to the master, asking the *MASTER* for an update message. Then, a message will be received from the *MASTER* in response to this *SLAVING* message in order to synchronize their data states. At this point, the slave node switchs its state to *SLAVE*.

During the *SLAVE* state, the only task of a node consists on listening for incoming messages from the master node. If no message is received from the master during a certain period of time, the slave node assumes the master is down, and transits to its *PROMOTION* state.

The primary node can save out *PING* messages, due to the existence of *UPDATE* messages: if a *PING* message is scheduled every K time units, and the next *PING* is scheduled to be sent at t_s and an *UPDATE* message is broadcast at t_i (with $i < s$) due to a change in the data-state of the primary node, then every slave will know about the master, through the *UPDATE* message, and the *PING* message scheduled at t_s can be delayed to t_{i+K}.

When a *SLAVE* node promotes its role to *MASTER*, it applies the data-state received from the old *MASTER*. The new *MASTER* continues the role of the old *MASTER* node from the point the old *MASTER* was before its failure.

In the presented algorithm, we have used the node identifier as the basis for the primary node election. In order to solve the identifier assignment to each node, we have used the MAC address of the private interface of each node as *node identifier*. Other similar assignment can be used to this end, honoring the order restriction of the identifiers:

$$\forall N_i, N_j, i \neq j : (i < j) \veebar (j > i)$$

3.4 Node Failures and Leader Promotion

A node failure has been presented as a complete failure of the node, producing its automatic stop. But this is not always true. A node can be temporally down, and

can try to continue later on. When the node is a slave, it only needs to restart its *SLAVE* state, asking the master its complete data-state. If the reconnected node was a master after its failure, there could be two primary nodes in the system. This is not a problem if we use MGP to notify a node that the incarnation view of the system is different than the current one. If MGP detects this situation, the MMM is notified, and then can start as a new node insertion in the system.

4 Other Minor Details

4.1 Network interfaces: MAC and IP configuration

In addition to the membership maintenance, the MMM is also used to configure the slave nodes in base to the existing master one (if any). When a node is first connected to the system (just when its state is *UNKNOWN*), it doesn't have any information about the IP's that are used as public IP and private IP of the HARL router. A new node is also unaware of which of its network interfaces are used as external and internal interface.

This information is requested by a new node in the system, by way of a new message sent to the master: *SAYIFTYPE*. When a node starts up first, it sends a broadcast message using it private network interface (the only interface actually known by the node). The primary node (in *MASTER* state) receives these messages, and sends a response message (of type *AUTOCONFIP*) to the original node (and using the internal interface) indicating the type of the interface used by the new node to send this *SAYIFTYPE* message (that is, INTERNAL). This method allows the new node to configure itself using the primary node as a "wall" for its messages. The *AUTOCONFIP* message sent by the primary node also contains the IP information needed by the new node to configure its interfaces.

4.2 HARL Guarantees

HARL has been designed to improve the scalability of the availability of the system, minimizing the point-to-point communication between the involved nodes, and avoiding the bandwidth consumption at the external and internal networks.

The recovery time of a single node failure is bounded to a well-known amount of time. This is determined mainly by the duration of the *PROMOTION* state in the MMM algorithm. The value of this time can be configured by tuning the system: it must be long enough to grant that only one node in the *PROMOTION* state ends successfully, no other node can end successfully its *PROMOTION* state in the same incarnation. This condition can be met taking into account the latency of the MGP transmissions over the private network segment. It is also necessary to adjust the *PING* message periodicity according to the value of the *PROMOTION* length.

The HARL router guarantees that after the recovery process, the established connections before the failure will remain active in the new incarnation.

The number of simultaneous failures allowed by the HARL system is $N - 1$ where N is the number of replicas in the HARL system.

5 Concluding remarks

We have presented HARL, a low cost, highly available router design based on Linux, using the primary-backup paradigm to recover from failures. HARL embodies a simple communication (MGP) and membership protocol (MMM) capable of keeping the state of the routing tables up-to-date in all the live replicas, while at the same time detecting failures of the primary node, and promoting one of the backups to the primary status. The checkpointed state in the replicas guarantees that existing connections are not closed as a result of a failure.

HARL has been implemented within Linux using loadable modules, with minimal changes to the kernel sources, which guarantees its adaptability to new kernel versions.

Hardware set-up for HARL requires only standard Linux boxes with an additional Ethernet network interface (although existing USB ports could also be used). The configuration process (adding and removing replicas) is totally automatic, needing only external intervention in initial router setup.

HARL recovers within a guaranteed maximum time which is related to the latency of the internal network used (10-base-2 Ethernet in our prototype). This time is low enough not to provoke closing down of existing connections.

References

1. Aguilera, Chen, and Toueg. Failure detection and consensus in the crash-recovery model. In *International Symposium on Distributed Computing, LNCS*, volume 12, 1998.
2. A. Avizienis. Fault tolerant systems. *IEEE Transactions on Computers*, December 1976.
3. M. Barborak, M Malek, and A Dahbura. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, 1993.
4. N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. *Distributed Systems*, chapter The primary backup approach, pages 199–216. Addison-Wesley, 2nd edition, 1993.
5. Om P. Damani, P. Emerald Chung, Yennun Huang, Chandra Kintala, and Yi-Min Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29(8–13):1019–1027, 1997.
6. K. Egevang and P. Francis. *RFC 1631: The IP network address translator (NAT)*, May 1994.
7. P. Galdámez F. D. Muñoz-Escoí, O. Gomis-Hilario and J. M. Bernabéu-Aubán. Hmm: A cluster membership service. In *EUROPAR 2001*. Springer-Verlag, ago 2001.
8. F.Baker. *RFC-1812: Requirements for IP Version 4 Routers*, June 1995.
9. P. Galdámez, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. High availability support in CORBA environments. In F. Plášil and K. G. Jeffery, editors, *24th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic*, volume 1338 of *LNCS*, pages 407–414. Springer Verlag, November 1997.
10. T. Hain. *RFC-2993: Architectural Implications of NAT*, November 2000.
11. S. Horman. Creating redundant linux servers. *4th Annual Linux Expo*, may 1998.
12. D.C. Plummer. *RFC 826: Ethernet Address Resolution Protocol*, November 1982.
13. L. Vepstas. *Linux Network Address Translation*, November 1998.

Localización de datos y reconfiguración en ALBADES

Juan S. Sendra Roig, José Bernabéu Aubán

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia, Camino de Vera s/n, 46027 Valencia
{jsendra, josep}@iti.upv.es

Abstract. Albades (ALmacenamiento BAsico Distribuido EStable) es una propuesta alternativa para el subsistema de almacenamiento de un cluster, que se plantea como una herramienta de exploración, experimentación e investigación. Uno de los aspectos básicos del sistema es la utilización de componentes estándar de bajo coste, explotando la economía de escala. En este contexto la escalabilidad y facilidad de reconfiguración son aspectos básicos del diseño. Para mejorar el nivel de escalabilidad y reducir el coste de reconfiguración (ej.- tras un fallo) se utiliza un esquema dinámico de localización descentralizada de los datos. Dicho esquema facilita la migración de datos y el equilibrado de carga, y permite reconfigurar el cluster sin necesidad de acuerdo atómico entre los nodos. El texto detalla el esquema de localización de datos y reconfiguración y muestra resultados obtenidos mediante simulación. Comparando con otros sistemas descentralizados, los primeros resultados muestran una ligera mejora en ausencia de fallos, y una mejora significativa al reconfigurar el sistema. La ventaja frente a otros esquemas aumenta con el número de nodos, mejorando significativamente la escalibidad del sistema.

Descripción del problema

Cada vez encontramos más aplicaciones que requieren acceso continuo a datos críticos. En dichas aplicaciones la interrupción del servicio tiene un elevado coste (aplicaciones bancarias, reserva de billetes aéreos, servicio ftp o web, etc.) o es simplemente inaceptable (ej.- apoyo al control aéreo, sistemas médicos, sistemas de correo de alta disponibilidad, etc.).

La técnica de clustering permite mantener el servicio activo y disponible de forma permanente. La replicación interna permite responder a los fallos reconfigurando dinámicamente el sistema sin interrumpir el servicio (gestión de fallos transparente al usuario). Con el software adecuado, el sistema puede soportar tanto fallos en dispositivos como fallos catastróficos que afectan a nodos completos.

Los nodos del cluster comparten el subsistema de almacenamiento; cuando se detecta el fallo de un nodo, otro nodo accede al estado salvaguardado en el almacenamiento compartido y asume sus tareas. El sistema de almacenamiento compartido es un recurso crítico, cuya disponibilidad condiciona la disponibilidad del sistema y cuya escalabilidad resulta crítica.

Nos centramos en clusters en los que cada nodo posee acceso íntegro al sistema de almacenamiento, y ejecuta una aplicación que implanta el equivalente a una base de datos consistente compartida entre los nodos:

- El acceso al sistema de almacenamiento utiliza una utilidad de gestión de reservas distribuida. El uso de un fichero mantiene una reserva sobre el mismo; si otro usuario requiere acceso al fichero, el gestor de reservas distribuido condiciona el acceso al tipo de reserva que existe actualmente sobre dicho fichero.
- Si falla un nodo, cualquiera puede tomar el relevo. Tras un fallo, el gestor de reservas distribuido realiza la asignación del fichero a otro nodo.

Este mecanismo permite reconfiguración muy rápida y garantiza buena escalabilidad.

Para optimizar la fiabilidad y escalabilidad el subsistema de almacenamiento, debemos distribuir entre los nodos no sólo los datos, sino también la responsabilidad sobre los mismos (sistemas serverless). Existen pocas propuestas en este línea, siendo xFS [xFS] la más conocida, y la práctica totalidad se plantean como sistemas monolíticos que explotan hardware especializado (ej.- esquema de red que soporta multicast, redes conmutadas de altas prestaciones, sistemas RAID replicados, etc.). En consecuencia:

- Su coste es elevado.- implica ámbito de aplicación reducido
- Son sistemas complejos.- dificulta su desarrollo, mantenimiento, y posterior modificación.

ALBADES: visión global

Albades (ALmacenamiento BAsico Distribuido EStable) es una propuesta alternativa para el subsistema de almacenamiento de un cluster. El cluster está formado por un conjunto de nodos homogéneos (cada nodo es un PC ejecutando el mismo sistema operativo, aunque puede tener dispositivos y características hardware diferentes) interconectados mediante una red rápida dedicada (ej.- gigabit ethernet).

A nivel funcional, Albades plantea los siguientes objetivos:

- Reducción del coste mediante la explotación de la economía de escala (utilización de componentes estándar de bajo coste), incidiendo especialmente en la escalabilidad
- Mejora en la flexibilidad y las prestaciones, mediante un diseño modular que facilita la incorporación de nuevas soluciones software y mejora el soporte para un mayor rango de aplicaciones (especialmente aplicaciones transaccionales, medios continuos, y servicio web)
- Alta disponibilidad e integridad
  - Evita puntos de fallo únicos, y limita la necesidad de acuerdo atómico entre los nodos
  - Ofrece garantías sobre la integridad de los datos
  - Optimiza el tiempo de recuperación del sistema
  - Ofrece garantías de atomicidad y durabilidad en el almacenamiento
- Escalabilidad
  - Evita estructuras de datos centralizadas

- Distribución de responsabilidades y datos entre los distintos nodos

Una de las contribuciones es el esquema dinámico de localización descentralizada de los datos, que también se explota para reconfigurar el cluster tras un fallo sin necesidad de acuerdo atómico entre los nodos.

El cluster define los mecanismos necesarios para facilitar la cooperación entre los nodos (y por tanto el paralelismo interno), ofreciendo simultáneamente tolerancia a fallos y mecanismos que facilitan la escalabilidad el sistema:

- La ejecución de un protocolo de pertenencia al cluster garantiza que todos los nodos detectan los fallos/recuperaciones en el mismo orden, y por tanto mantienen una visión consistente del estado del sistema
- La configuración puede variar (ej.- ante fallo nodo) de forma dinámica, automática, y transparente (a usuarios y aplicaciones)
- Distribución de datos y responsabilidades.- los datos y metadatos se distribuyen entre los distintos nodos, evitando cuellos de botella y puntos de fallo únicos, y mejorando la escalabilidad. Este esquema complica la localización de los datos, y ha conducido al desarrollo de un nuevo esquema de localización de datos y reconfiguración del sistema.

La complejidad del sistema conduce a un desarrollo gradual mediante simulación y/o prototipado de los aspectos más novedosos, y posterior integración de funciones en un sistema real. Actualmente se están utilizando simulaciones para validar las características del esquema propuesto.

Localización de la información

El esquema de localización de la información se define como una aplicación (mapping) entre un objeto (agrupación de información) y el nodo donde reside. Se trata de una pieza vital en los mecanismos de recuperación, almacenamiento en disco, y mantenimiento de la coherencia entre cachés.

Podemos aplicar esquemas centralizados o distribuidos tanto a datos como a metadatos (ej.- información necesaria para localizar los datos y garantizar la semántica de compartición de los mismos). Analizamos las distintas posibilidades.

Centralización de datos y metadatos (Ej NFS).- Utiliza un servidor donde se centralizan tanto los datos como los metadatos. Cuando un cliente desea acceder a un objeto remite la petición al servidor, que devuelve la correspondiente respuesta

Es el esquema más simple. El servidor constituye un punto de fallo único y se convierte en un cuello de botella, penalizando la tolerancia a fallos, las prestaciones, y la escalabilidad del sistema..

Centralización de metadatos, distribución de datos (Ej.- Sprite).- Mantiene la centralización de metadatos, distribuyendo los datos entre los distintos nodos. Cuando un cliente desea acceder a un objeto, remite una petición al servidor central, el cual determina la ubicación de los datos y envía un mensaje al nodo que posee el objeto para que éste remita directamente el objeto al cliente que lo solicitó.

Este esquema descarga parcialmente al servidor central (mejora las prestaciones y la escalabilidad), pero a cambio cada acceso no local requiere un mensaje adicional, y el servidor sigue siendo un punto de fallo único y un cuello de botella potencial.

Distribución de datos y metadatos.- Cada nodo registra la ubicación de una parte de los objetos del sistema, dirigiendo las peticiones de los clientes a las posiciones correctas y coordinando el acceso de múltiples clientes a los datos de los que se responsabiliza.

A priori mejora la tolerancia a fallos y la escalabilidad, pero requiere que cada nodo posea información suficiente para localizar directamente los datos o averiguar qué nodo conoce dicha ubicación, y exigimos que no incremente el número de mensajes necesarios respecto de un sistema que centraliza los metadatos.

Entre los sistemas que distribuyen datos y metadatos podemos realizar una clasificación según distintos criterios:

- Mecanismo de partición.- cada nodo se responsabiliza de los metadatos correspondientes a un conjunto de ficheros (ej.- directorio) o bien de los metadatos correspondientes a un conjunto de objetos (rangos dentro de ficheros)
- Esquema de localización.- Se utilizan sistemas basados en hash o bien una distribución jerárquica (árboles distribuidos)
  - Hash.- el mapping entre nodos y objetos se calcula mediante una función matemática. La mayor parte de los sistemas utilizan funciones hash estáticas, pero resulta posible diseñar funciones hash dinámicas (ej.- hashing lineal dinámico) o soluciones basadas en un conocimiento aproximado (LH* [LH*]).
  - distribución jerárquica (árboles distribuidos). En este momento su uso es limitado (ej máquina KSR1), por los inconvenientes que presenta (el elemento raiz puede convertirse en un cuello de botella y condicionar la disponibilidad, y el número de mensajes depende de la profundidad de la jerarquía), pero recientemente se han planteado nuevas propuestas que pueden resultar interesantes (ej.- HB-tree [HBtree]).
- Tipo de mapping.- distinguimos entre mapping fijo (cuya reconfiguración requiere la parada temporal del sistema, acuerdo atómico entre los nodos, y difusión del nuevo mapping a los distintos nodos) y mapping flexible (permite reconfigurar el mapping sin necesidad de acuerdo atómico).

La implementación del mapping fijo es mucho más simple (ej.- función hash o directorio totalmente replicado), pero sólo se justifica cuando no se considera la posibilidad de fallo o se dispone de hardware especial de difusión que facilita el acuerdo y actualización atómicos. Esta solución se utiliza en algunos sistemas de ficheros (ej.- Vesta), la mayor parte de los sistemas de memoria distribuida (DSM), y sistemas de mantenimiento de la consistencia en cachés multiprocesador.

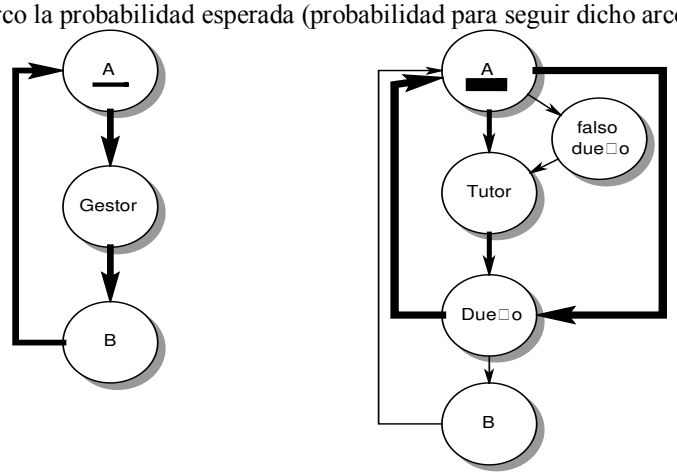
Pocos sistemas de ficheros distribuyen totalmente los datos y la responsabilidad sobre los mismos, y en su mayor parte utilizan mapping fijo. La excepción es xFS, que a pesar de utilizar a priori un esquema de localización basado en una función hash estática, consigue cierta flexibilidad al superponer un nivel de indirección (vector de indirección). Algunos sistemas de memoria virtual utilizan un esquema similar (ej.- Feeley).

Nuestra propuesta persigue distribución total e independencia de ubicación, pero intenta facilitar la migración de granularidad fina y agilizar la reconfiguración del sistema. Utilizamos xFS como base de comparación; el objetivo será diseñar un esquema que mantenga la eficiencia de xFS pero ofrezca mayor flexibilidad (mapping flexsible).

Localización de datos en Albades.- Cada nodo expresa su visión del resto del sistema en forma de pistas (ej.- información obtenida en un acceso anterior a un fichero) que sirven para optimizar el acceso. Las pistas pierden fiabilidad con el tiempo y llegan a caducar, por lo que existe un mecanismo para localizar los datos en ausencia de pistas. El objetivo es utilizar las pistas para:

- Reducir el número global de mensajes.- las pistas sobre un elemento son más útiles (acceso óptimo) cuanto mayor es la frecuencia de acceso al mismo. El caso peor (pista no válida) es más costoso, pero menos frecuente
- Facilitar la reconfiguración.- tras un fallo se puede proseguir de inmediato sin esperar a que los distintos nodos acuerden una visión global del sistema.

La siguiente figura ilustra la localización de datos en xFS y Albades. Los diagramas ilustran conjuntamente los distintos supuestos, indicando mediante el grosor de cada arco la probabilidad esperada (probabilidad para seguir dicho arco).



En Albades aparecen varios caminos alternativos de distinto coste: el objetivo es que la mayor parte de las peticiones sigan el camino óptimo, y el caso peor sea relativamente infrecuente.

En ambos sistemas es posible el acceso local (0 mensajes), aunque la facilidad para migrar ficheros individuales en Albades debería mejorar la localidad (mayor peso en los accesos locales en la figura).

En caso de acceso remoto (desde nodo A), ambos sistemas aplican una función hash sobre el identificador interno del fichero:

- En xFS la función hash devuelve un bucket, y a través de una tabla buckets-nodos totalmente replicada se obtiene un nodo que se denomina nodo gestor (gestor= mapa[hash(idFichero)]).

- Lectura.- Cada gestor mantiene la lista de clientes que poseen copias de algunos de sus bloques, y reenvía las peticiones de lectura a otros clientes que tamponan los bloques (ej.- a B), los cuales devuelven la respuesta directamente a A
- Escritura.- Para obtener la propiedad de un bloque, A envía un mensaje al gestor, el cual invalida las restantes copias del bloque (para mantener la coherencia), actualiza la propiedad del mismo, y le devuelve el permiso a A.

- En Albades las responsabilidades del gestor se reparten entre un nodo tutor y el nodo dueño. El nodo tutor conoce la identidad del actual dueño del fichero (nivel de indirección), y el nodo dueño mantiene la lista de clientes y mantiene la coherencia de la caché para ese fichero.
  - Lectura.- A partir del identificador del fichero se calcula el slot, y mediante una tabla de slots el nodo tutor, que a su vez redirige el mensaje al dueño (si coinciden tutor y dueño la respuesta es directa –no se ha representado en la figura-), y éste devuelve respuesta a A (o bien, si está sobrecargado, redirige el mensaje a un cliente B que mantenga copia de los datos, y B los envia hacia A).
  - Escritura.- Significa la obtención de la propiedad y la invalidación de las restantes copias del fichero. A envía un mensaje al dueño (usando la misma estrategia definida para la lectura), el cual invalida las restantes copias del bloque, avisa al nodo tutor, y transfiere la propiedad (A pasa a ser el nuevo dueño)

- Además cada nodo mantiene pistas (último dueño conocido para cada fichero):
  - Si la pista es correcta, acceso directo al dueño
  - Si no existe pista (Ej.- primer acceso o la pista ha caducado), acceso al tutor
  - Si la pista es incorrecta se accede a un falso dueño, que a su vez redirige al tutor

Equilibrado de carga y reconfiguración

El problema de la localización se complica al introducir otros dos requisitos: por razones de fiabilidad y disponibilidad, los datos pueden estar replicados en varios nodos (mapping 1:n), y dicho mapping debe ser flexible (ej.- para facilitar el equilibrado de carga o la reconfiguración tras un fallo). Cualquier objeto puede almacenarse en cualquier nodo, y la ubicación puede variar con el tiempo

Ilustramos nuestra propuesta comparando con xFS.

Reconfiguración en xFS

xFS define un conjunto de buckets (fragmentos del sistema de ficheros) y un mapa de gestores (mapping bucket-nodo) totalmente replicado (es una tabla relativamente pequeña y cambia con poca frecuencia). El sistema aplica una función hash sobre el identificador interno del fichero para determinar el bucket, y consulta el mapa de gestores para determinar qué máquina gestiona dicho bucket. Existen muchos más

buckets que nodos, de forma que se puedan migrar buckets entre nodos para equilibrar la carga o reconfigurar el cluster.

El mapa de gestores limita los cambios en los gestores a partes aisladas de su índice (menos interrupciones, y los cambios que introduce un gestor no interfieren con los esfuerzos de distribución de carga y localidead que hacen otros). Existen dos posibilidades de migración:

- Ajuste fino.- manipular los identificadores internos de los ficheros (elección del identificador durante la creación del fichero –inflexible-, o modificación a posteriori –requiere acuerdo atómico entre los nodos-)
- Ajuste grueso.- modificar el mapa de gestores. Se reduce la localidad, porque la reasignación se realiza por grupos de ficheros. Para facilitar la migración y reconfiguración el número de buckets debe ser al menos 10 veces el de nodos.

Para reconfigurar el sistema tras un fallo, se sigue el siguiente esquema:

- Todos los gestores activos participan en un algoritmo de consenso global para acordar un nuevo mapa de gestores.
- Un algoritmo de consenso identifica a todas las máquinas que actuarán como gestores y elige un lider entre ellas
- El lider crea un nuevo mapa, donde asigna el mismo número de entradas a cada gestor activo
- El lider difunde la tabla a los restantes gestores

Los gestores recuperan el estado de consistencia de sus ficheros preguntando a los clientes que mantenían copias de los mismos.

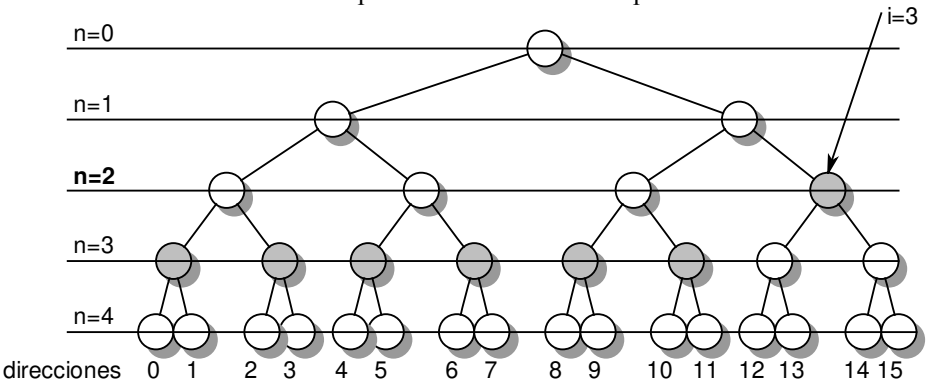
Reconfiguración en Albades

En Albades la propiedad de un fichero corresponde al nodo que utiliza más activamente dicho fichero, y dicha propiedad puede migrar (soprota migración a nivel de ficheros individuales, o sea de granularidad fina). Hablamos de slots (concpeto idéntico al de buckets de xFS), pero permitimos un número variable de slots, y apenas requiere doble slots que nodos:

- También existe una tabla slots-nodos, pero mucho más reducida
- Cuando se detecta la baja o alta de un slot, todos los nodos siguen el mismo algoritmo (descrito posteriormente) para rediseñar sus mapas de forma autónoma, con lo que no se requiere acuerdo atómico

En Albades, cuando un nodo detecta el fallo del nodo i rehace su mapa slots/nodos de forma autónoma, crea una lista de los ficheros de su propiedad de los que i era tutor, y envía la lista al nuevo tutor (migración de slot). Con este esquema no existe periodo de inactividad tras un fallo. Si el cliente todavía no ha detectado el fallo, no ha aplicado el algoritmo de reconfiguración, y calculará un tutor incorrecto, pero cada intento fallido reduce el desfase y mejora la probabilidad de seguir luego el camino óptimo.

Supongamos un arbol binario quasi-equilibrado donde todas las hojas están a nivel n o n+1, y se completa por niveles de izquierda a derecha. Cada par de nodos de nivel n+1 ha surgido tras la expansión de uno de los nodos de nivel n (la expansión también procede de izquierda a derecha). Dicho árbol puede definirse en base a los valores n (nivel a expandir) e i (siguiente elemento del nivel n a expandir). Cada nodo representa un rango de direcciones (o de identificadores), de forma que una vez definido el árbol una función simple obtiene el nodo correspondiente a la dirección i.



Los nodos sombreados representan 7 slots (n=2, i=3). Slots = $2^n + i$

El esquema asume que todos los nodos observan los fallos/recuperaciones en el mismo orden, y aplican los mismos algoritmos como respuesta. Supongamos definidas las siguientes rutinas:

ExpandeNodo: se migra la segunda mitad de su espacio de direcciones a un nuevo nodo (el nodo expandido ...x queda como ...0x, y el nuevo como ...1x)
ColapsaNodo: se une su espacio de direcciones con el de otro nodo que ya existe. Se pueden unir nodos con el mismo patrón de bits antes de la x excepto el último bit, que en un nood debe ser 0 y en el otro 1
SubeNivel.- i vuelve a 0 y se incrementa n
BajaNivel.- se decrementa n, e i pasa a valer 2^n –1

Algoritmo de inserción de nodos (recuperación nodo o expansión cluster)

Se expande el nodo i y se incrementa el valor de i. Si i>=2^n, sube nivel
La tabla de slots se limita a dividir el espacio de identificadores en partes iguales (si i=0, 2^n slots, y en caso contrario, 2^(n+1) slots) y reflejar qué nodo se responsabiliza de cada fracción del espacio de identificadores

N	I	Tabla slots	Direcciones responsabilidad de cada nodo							
			0	1	2	3	4	5	6	7
0	0	0	X							
1	0	01	0x	1x						
1	1	0211	00x	1x	01x					
2	0	0213	00x	10x	01x	11x				

2	1	04221133	000x	10x	01x	11x	001x			
2	2	04251133	000x	100x	01x	11x	001x	101x		
2	3	04251633	000x	100x	010x	11x	001x	101x	011x	
3	0	04251637	000x	100x	010x	110x	001x	101x	011x	111x

Algoritmo de eliminación de nodos (tras fallo)

Sea A el nodo que ha fallado y U el último nodo (aparecerá como último del nivel n+1 –o del nivel n, si no hay nodos en n+1-):

- si A distinto de U, migramos la responsabilidad de A hasta U
- colapsamos U y decrementamos el valor de i. Si i=0, se reduce a la mitad el número de slots. Si i<0, baja nivel

Falla	N	I	Tabla slots	Direcciones responsabilidad de cada nodo							
				0	1	2	3	4	5	6	7
	3	0	04251637	000x	100x	010x	110x	001x	101x	011x	111x
2	2	3	04761533	000x	100x	--	11x	001x	101x	011x	010x
5	2	2	04761133	000x	10x		11x	001x	--	011x	010x
1	2	1	04667733	000x	--		11x	001x		01x	10x
3	2	0	0674	00x			--	11x		01x	10x
6	1	1	0477	00x				01x		--	1x
7	1	0	04	0x				1x			--
0	0	0	0	--				X			

Resultados de la simulación

Se ha desarrollado software de simulación ad-hoc para comparar las prestaciones del sistema definido en xFS y el sistema propuesto en Albades.

Se han realizado cinco pruebas con cada combinación de parámetros, mostrando en los gráficos el valor promediado de las distintas pruebas, y eliminando en cada caso el transitorio previo a la estabilización del sistema.

Se han calculado los costes como número de mensajes (mensaje corto de control), y transferencias (mensaje largo para transmisión de datos). Este esquema permite estimar las prestaciones utilizando distintas redes de interconexión (ej.- el coste de los mensajes cortos depende esencialemente de la latencia de la red y la posibilidad de simultanear mensajes, mientras que las transferencias dependen en mayor medida del ancho de banda).

Las siguientes tablas ilustran las distinas situaciones que pueden presentarse ante cada operación de lectura y escritura en cada uno de los sistemas, y muestran el coste asociado a cada situación.

xFS (lectura)			
Hay copia local	Hay copia en gestor	Mensajes	Transf.
Si		0	0
No	Si	1	1
	No	2	1

xFS (escritura)				
Hay copia local	Tiene la propiedad	Hay copia en gestor	Mensajes	Transf.
Si	Si		0	0
	No		2	0
No		Si (*)	1	1
		No	2	1

albades (lectura)					
Hay copia local	Hay pista	Pista ok	Hay copia en tutor	Mensajes	Transf.
Si				0	0
No	Si	Si		1	1
		No	Si	2	1
			No	3	1
	No		Si (*)	1	1
			No	2	1

albades (escritura)						
Hay copia local	Propiedad	Hay pista	Pista ok	Tutor=dueño	Mensajes	Transf.
Si	Si				0	0
	No	Si	Si		2	0
			No	Si	3	0
				No	4	0
		No		Si	2	0
				No	3	0
No	No	Si	Si		1	1
			No	Si	2	1
				No	3	1
		No		Si	1	1
				No	2	1

Algunas de las situaciones planteadas en las tablas anteriores han resultado ser poco probables y por tanto poco significativas para interpretar los resultados finales. Para simplificar las gráficas se han eliminado de las mismas las series de datos correspondientes a situaciones poco significativas (con valores consistentes por debajo del 2% en todas las combinaciones de parámetros). Los casos poco significativos corresponden a las filas marcadas con (*).

Modelo de simulación

Para construir un modelo más exacto, se han introducido los siguientes elementos:

- Cada nodo accede únicamente a una parte de los ficheros del sistema, tanto para lectura (conjunto de lectura de dicho nodo) como para escritura (conjunto de escritura). Existen solapes entre conjuntos de lectura y escritura de distintos nodos, pero también pueden aparecer ficheros locales.
- El acceso a distintos ficheros no es equiprobable, para modelar ficheros con mayor y menor tasa de acceso

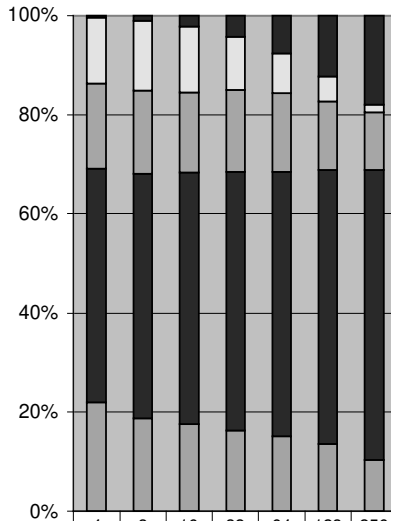
- Se ha modelado la localidad de acceso mediante una ventana deslizante (modelo working-set).

Se han realizado gran cantidad de pruebas variando distintos parámetros, a fin de observar la sensibilidad de los resultados a la variación de distintos aspectos del modelo. Las figuras presentadas corresponden los siguientes valores:

- 100 ficheros por nodo como promedio.
- Tasa de lectura del 70% (30% para escrituras)
- Caché en cada nodo.- política LRU, capacidad=4
- Talla del conjunto de lectura cuatro veces mayor que la del conjunto de escritura
- Ventana de acceso (working set) = 40

La primeras gráfica desglosa para cada caso el porcentaje de accesos de cada tipo, y la evolución de dichos porcentajes al variar el número de nodos. La segunda muestra el número medio de mensajes necesarios por acceso, y la tercera el número medio de transferencias por acceso. Un mensaje corresponde a un mensaje corto (ej.- solicitud), y una transferencia a un mensaje largo (conteniendo los datos del fichero).

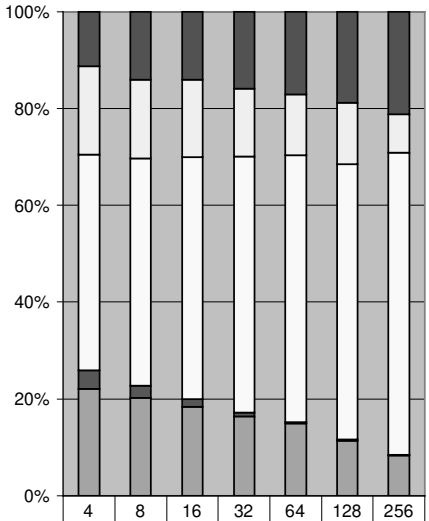
Localización datos (Albades)



	4	8	16	32	64	128	256
escritura (dueño)	0,4	1	2,2	4,3	7,6	12,3	17,9
escritura (tutor)	12,8	13,8	12,9	10,5	8	5	1,6
escritura local	16,6	16,4	15,8	16,4	15,8	13,7	11,6
lectura remota	45,4	48,2	49,4	51,4	53	55,2	58,4
lectura local	21,2	18,3	17,2	16,1	15	13,6	10,4

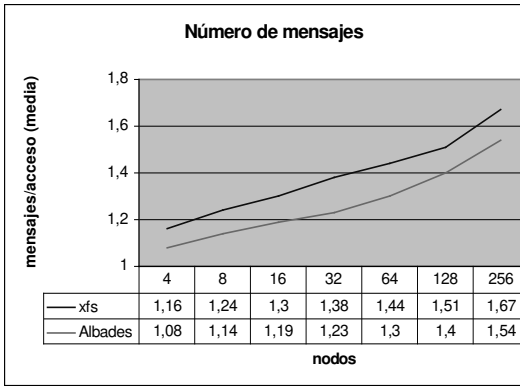
nodos

Localización datos (xFS)

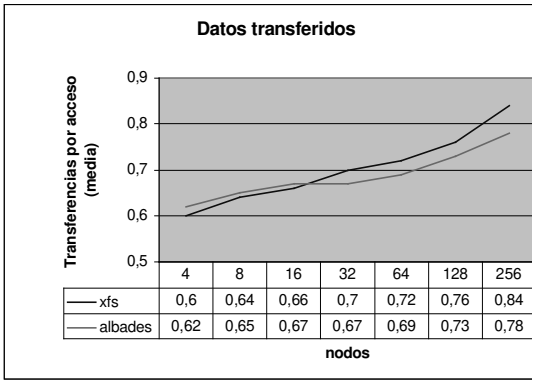


	4	8	16	32	64	128	256
escritura remota	11,2	13,9	13,9	15,8	17	18,8	21,2
escritura local	17,9	16,2	15,9	14	12,6	12,7	8
lectura remota	43,9	46,5	49,6	52,7	55	56,8	62,3
lectura desde gestor	3,7	2,5	1,6	0,7	0,3	0,2	0,1
lectura local	21,8	20,1	18,2	16,4	14,9	11,4	8,4

nodos



	4	8	16	32	64	128	256
xfs	1,16	1,24	1,3	1,38	1,44	1,51	1,67
Albades	1,08	1,14	1,19	1,23	1,3	1,4	1,54



	4	8	16	32	64	128	256
xfs	0,6	0,64	0,66	0,7	0,72	0,76	0,84
albades	0,62	0,65	0,67	0,67	0,69	0,73	0,78

Los datos no muestran diferencias significativas entre ambos modelos durante el procesamiento normal, con cierta ventaja para albades cuando crece el número de nodos.

Los valores absolutos que aparecen en las tablas son muy sensibles a la localidad de las referencias generadas. Sin embargo, generando de forma estática distintos patrones de referencia aleatorios no se han encontrado diferencias relativas significativas entre ambos métodos.

La variación de la proporción entre lecturas y escrituras no modifica de forma significativa las prestaciones relativas de ambos métodos.

Futuros trabajos

Se ha empezado a simular una carga variable con el tiempo, en la que albades resulta beneficiado debido a la facilidad para migrar la propiedad fichero a fichero. Se espera obtener cifras contrastadas en breve plazo.

Actualmente no se está calculando la carga soportada individualmente por cada uno de los nodos (excepto a nivel de caché, el resto de los recursos de cada nodo se suponen infinitos). Cuando se mejore el software de simulación podrán analizarse distintas estrategias de equilibrado de carga mediante migración de granularidad fina, y su repercusión en las prestaciones.

Hay dos técnicas que en un sistema real permiten mejorar las prestaciones de ambos sistemas, y que por el momento no se han incluido en la simulación:

- En xfs se pueden generar explicitamente un número de fichero durante la creación que permita asociar el resultado de la función hash sobre dicho fichero al nodo que lo crea (de forma que gestor y dueño de los datos coincida). Sin embargo, dicha asociación no puede migrar con el tiempo (ej.- no soporta equilibrado de carga de granularidad fina)
- En albades puede utilizarse piggybacking para asociar a mensajes normales información que ayude a actualizar las pistas. Otra posibilidad complementaria es establecer un periodo de caducidad para las pistas, de forma que se limiten las consecuencias de seguir una pista falsa

El principal aspecto pendiente de simulación es la respuesta ante cambios en la composición del cluster (reconfiguración tras añadir/eliminar nodos). En dichas situaciones nuestra solución no requiere acuerdo atómico entre los nodos, y permite que el sistema siga funcionando normalmente. La única consecuencia debería ser un incremento temporal de accesos costosos debidos a pistas erróneas

Conclusiones

El texto presenta una nueva propuesta para el diseño del subsistema de almacenamiento de un cluster. Se desea reforzar las caracteristicas de disponibilidad y escalabilidad sin recurrir a hardware especializado, para lo cual distribuye totalmente tanto los datos como la responsabilidad sobre los mismos, utiliza técnicas de replicación y dispersión de datos, y desarrolla un algoritmo de localización de información y reconfiguración que no requiere acuerdo atómico entre nodos.

El uso de una variante de hashing lineal distribuido para localizar los distintos componentes del sistema de almacenamiento evita cuellos de botella y puntos de fallo únicos (mejora la escalabilidad, y los cambios no requieren acuerdo atómico entre los nodos). Estas características son especialmente interesantes en un diseño que pretende explotar el paralelismo para mitigar la carencia de hardware especializado.

Para validar el diseño se ha recurrido a desarrollar software de simulación especifico. Los primeros resultados demuestran que incluso en el caso normal (sin reconfigurar el cluster, ni aplicar equilibrado de carga) el esquema propuesto se comporta ligeramente mejor que el esquema utilizado en xFS. La ventaja comparativa es mayor cuando se incrementa el número de nodos.

Posteriores simulaciones pretenden incluir nuevos aspectos (ej.- algoritmos para equilibrado de carga) y especialmente evaluar el coste de reconfiguración del cluster. El diseño del nuevo esquema debería suponer una importante mejora de prestaciones en esos casos, debido a la flexibilidad y capacidad de adaptación incremental.

Referencias

1. [dtfs] Eliezer Levy, Abraham Silberschatz, “Distributed File Systems: Concepts and Examples”, ACM 1990
2. [jornadas] Juan S. Sendra, José Bernabeu, “Ficlus: un nueva propuesta de almacenamiento en log” VII jornadas de Concurrencia, Gandía 1999
3. [LH*] Witold Litwin, Marie-Anne Neimat, Donovan A. Schneider, “LH*: a scalable, distributed data structure”, ACM Transactions on database systems, 21(4), pp 480.525, dic 1996
4. [lham] Peter Muth, Patrick E. O’Neil, Achim Pick, Gerhard Weikum, “The LHAM Log-Structured history data access method”, VLDB journal, Vol 8 num. 3-4 pp 191-221, feb 2000
5. [lsmt] P. O’Neil, E. Cheng, D. Gawlick, E.O’Neil, “The Log-structured Merge-Tree (LSMTree)”, Uni Mass/Boston Math & CS Dept. Technical Report 921, Julio 1992

6. [multicast] Kenneth P. Birman, Thomas A. Joseph, "Reliable communication in the presence of failures", Technical Report, Cornell University, Computer Science department, num TR85-694, p 25, Julio 1985
7. [PACA] T. Cortes, S.Girona and J. Labarta. PACA: A cooperative File System Cache for Pararell Machines". In 2nd International Euro-Par Conference (Euro-Par'96), pages 477-486, 1996, Lecture Notes in Computer Science1123
8. [RVM] Michael J. Feeley, Jeffrey S. Chase, Vivek R. Narasayya, Henry M. Levy, "Integrating coherency and recoverability in Distributed Systems", Proceedings of the 1st Symposium on Operating Systems Design and Implementation, pp 215-227, 1994
9. [tsbt] David B. Lomet, Betty Salzberg, "The performance of a multiversion access method", Proceedings of the 1990 ACM SIGMOD International Conference On Management of Data, Atlantic City, NJ, May 1990
10. [Vesta] Sheng-Yang Chiu, Roy Levin, "The Vesta Repository: A file system extension for software Development", Technical report, Digital Equipment Corporation, Systems Research Center, num 107, jun 1993
11. [vmmc] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos Damianakis, Kai Li, "VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication", Hot Interconnects V, Agosto 1997
12. [wal] A. Knaff, P. Dechamboux, "Reliable support for a persistent distributed shared memory", 17th internarional conference on Distributed Computing Systems (17th IDCS'97), pp 68-77, mayo 1997
13. [wobt] Malcolm C. Easton, "Key-Sequence Data Sets on Inedible Storage", IBM Journal of Research and Development 30 (3), 230-241, 1986
14. [xFS] 2.-T.E. Anderson, M.H.Dalhin, J.M. Neefe, D.A. Pattterson, D.S. Roselli, and R.Y. Wang, "Serverless Network File Systems", ACM Transactions on Computer Systems, 14(1): 41-79
15. Asit Dan, Philip S. Yu, Anant Jhingran, "Recovery analisys of data sharing systems under deferred dirty page propagation policies", IEEE transactions on pararell and distributed systems, vol 8 no. 7 julio 1997

Analysis of the MPEG-2 Encoder Algorithm with Timed-Arc Petri Nets *

Valentín Valero, Fernando L. Pelayo, Fernando Cuartero, and Diego Cazorla

Departamento de Informática
Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha
Campus Universitario s/n. 02071. Albacete, SPAIN
{ valentin, fpelayo, fernando, dcazorla }@info-ab.uclm.es

Abstract. In this paper we analyse a parallel version of the MPEG algorithm for video encoding, by using Timed-Arc Petri nets. These are a timed extension of Petri nets, in which tokens have an age, and arcs connecting places with transitions are labelled with a time interval, thus establishing a restriction for the tokens that must be used in order to fire the transitions. With this model it is possible that some tokens become dead, in the sense that they cannot be used for the firing of any transitions in the future, and then we exploit this fact in order to extend the classical notion of safeness, thus defining the soft-safeness of a Timed-Arc Petri net. Then, we analyse a version of the MPEG–2 encoder, modelling it with a MTAPN. The main consequence that we can obtain from the model is that some parts of the algorithm can be made in parallel, and thus, provided that we have several processors, we can improve the performance of the encoder significatively.

1 Introduction

A survey of the different approaches to introduce time in Petri nets is presented in [5]. We can identify a first group of models, which assign time delays to transitions, either using a fixed and deterministic value [11, 12] or choosing it from a probability distribution [3]. Other models use time intervals to establish the enabling times of transitions [10]. Finally, we have also some models that introduce time on tokens [2, 4, 13]. In such a case tokens become classified into two different classes: available and unavailable ones. Available tokens are those that can be immediately used for firing a transition, while unavailable cannot. We have to wait for a certain period of time for these tokens to become available, although it is also possible for a token to remain unavailable forever (such tokens are said to be *dead*). More recently, Cerone and Maggiolo-Schettini [6] have defined a very general model (statically timed Petri nets), where timing constraints are intervals statically associated with places, transitions and arcs.

* This work has been supported by the CICYT project "Performance Evaluation of Distributed Systems", TIC2000-0701-C02-02.

Thus, models with timing constraints attached only to places, transitions or arcs can be obtained by considering particular subclasses of this general framework.

Timed-Arc Petri nets [1, 4, 7, 8, 13, 14] are a timed extension of Petri nets in which tokens have associated a non-negative real value indicating the elapsed time from its creation (*its age*), and arcs from places to transitions are also labelled by time intervals, which establish restrictions on the age of the tokens that can be used to fire the adjacent transitions. As a consequence of these restrictions some tokens may become *dead*, because they will be never available, since they are too old to fire any transitions in the future. The interpretation and use of Timed-Arcs Petri nets can be obtained from a collection of processes interacting with one another according to a rendez-vous mechanism. Each process may execute either local actions or synchronization ones. Local actions are those that the process may execute without cooperation from another process, and thus in the Petri net model of the whole system they would appear as transitions with a single precondition place, while synchronization actions would have several precondition places, which correspond to the states at which each one of the involved processes is ready to execute the action. Then, each time interval establishes some timing restrictions related to a particular process (for instance the time that a local processing may require). In consequence, the firing of a synchronization action can be done in a time window, which depends on the age of the tokens on its precondition places.

Therefore, Timed-Arc Petri nets are a very appropriate model for the description of concurrent systems with time restrictions, such as manufacturing systems, real-time systems, process control, workflow systems, etc. In this paper we show the use of TAPNs for the modeling of safe concurrent systems. But we also exploit the fact that some tokens may become dead in order to extend the classical notion of safeness, defining the *soft-safeness* of a TAPN. As illustration, we model the MPEG-2 Video Encoder by using TAPNs. The ISO/IEC 13818–2 standard [9], commonly known as MPEG–2, is a standard intended for a wide range of applications, including Video–on–Demand (VoD), High Definition TV (HDTV) and video communications using broadband networks.

The MPEG standards were designed with these two requirements:

- The need for a high compression, which is achieved by exploiting both spatial and temporal redundancies within an image sequence.
- The need for random access capability, which is obtained by considering a special kind of pictures (I pictures), which are encoded with no reference to other frames, only exploiting the spatial correlation in a frame.

This paper shows the specification and analysis of a singular version of parallel MPEG–2, in which instead of distribute the data among the processors by either distributing different partitions of the same frame (spatial parallelism) or different GoPs to the various processors (temporal parallelism), as usual, we will benefit from the intrinsic parallelism that we find in the MTAPN modelling this encoder for each GoP, and thus some improvements for the encoder can be proposed as a consequence of this.

The results so obtained improve significatively the performance of the standard MPEG–2, and we have studied the performance we can get both with two and ten processors.

The paper is structured as follows. In Section 2 we present timed-arc Petri nets and their semantics, and we also define the notion of *soft-safeness*. In Section 3 we show how to construct a state graph for bounded MTPANs, and how to extend this construction for soft-safe MTPANs. In Section 4 we describe how the MPEG-2 encoder works and we present a MTAPN that models this algorithm. Finally, the analysis of the algorithm and some conclusions are presented in Section 5.

2 Timed-Arc Petri Nets

We deal with timed-arc Petri nets, which have their tokens annotated with an age (a real value indicating the elapsed time from its creation) and arcs connecting places with transitions have associated a time interval, which limits the age of the tokens that must be consumed to fire the adjacent transition.

However, a transition is not forced to be fired when all its preconditions contain tokens with an adequate age, and the same is true even if the age of any of these tokens is about to expire. More in general, in the model we consider[1] there is not any kind of urgency, what we can interpret in the sense that the model is *reactive*, as transitions will be only fired when the environment requires it. But then, it can be the case that the external context may lose the ability to fire a transition if some needed tokens become too old. Even more, it is possible that some tokens become *dead*, which means definitely useless because their increasing age will not allow in the future the firing of any of their postcondition transitions.

2.1 Definitions

Definition 1. (Timed-arc Petri nets)
We define a timed-arc Petri net (TAPN) as a tuple [2] $N = (P, T, F, times)$, where P is a finite set of *places*, T is a finite set of transitions ($P \cap T = \emptyset$), F is the *flow relation* ($F \subseteq (P \times T) \cup (T \times P)$), and *times* is a function that associates a closed time interval to each arc (p, t) in F, i.e.: $times : F|_{P \times T} \longrightarrow \mathbb{R}_0^+ \times (\mathbb{R}_0^+ \cup \{\infty\})$.

When $times(p, t) = [x_1, x_2]$ we write $\pi_i(p, t)$ to denote x_i, for $i = 1, 2$.

As we previously mentioned, tokens are annotated with real values, so markings are defined by means of multisets on $\mathbb{R}_0^+$. More exactly, a marking M is a function:

$$M : P \longrightarrow \mathcal{B}(\mathbb{R}_0^+)$$

[1] Other proposals of Timed-arc Petri nets [8] enforce the firing of transitions with an earliest and maximal firing rule.

[2] We consider only arcs with weight 1 to simplify some definitions, but the extension to general arcs with greater weights is straightforward.

where $\mathcal{B}(\mathbb{R}_0^+)$ denotes the set of finite multisets of non-negative real numbers. Thus, as usual, each place is annotated with a certain number of tokens, but each one of them has associated a non-negative real number (its *age*). We will denote the set of markings of N by $\mathcal{M}(N)$, and using classical set notation, we will denote the number of tokens on a place p by $|M(p)|$.

As initial markings we only allow markings M such that for all p in P, and any $x > 0$ we have $M(p)(x) = 0$ (i.e., the *initial age* of any token is 0). Then, we define *marked timed-arc Petri nets* (MTAPN) as pairs (N, M), where N is a timed-arc Petri net, and M is an initial marking on it. As usual, from this initial marking we will obtain new markings, as the net evolves, either by firing transitions, or by time elapsing. In consequence, given a non-zero marking, even if we do not fire any transitions at all, starting from this marking we get an infinite reachability set of markings, due to the token aging.

A timed-arc Petri net with an arbitrary marking can be graphically represented by extending the usual representation of P/T nets with the corresponding time information. In particular we will use the age of each token to represent it. Therefore, MTAPNs have initially a finite collection of zero values labelling each place. □

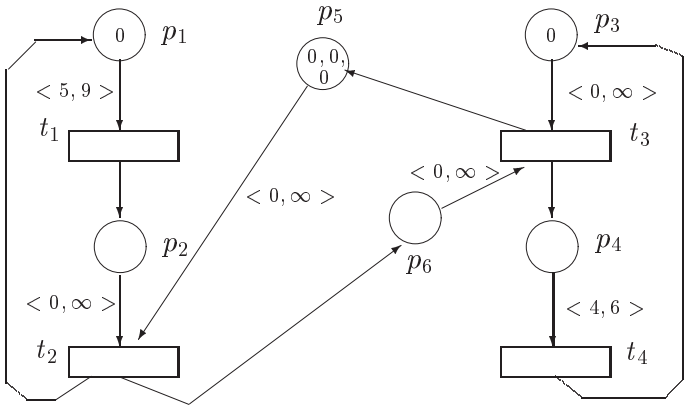


Fig. 1. Timed-arc Petri net modeling the PC-problem

In Fig. 1 we show a MTAPN modeling a producer/consumer system, where we have represented by transition t_1 the action corresponding to the manufacturing process of the producer, which takes between 5 and 9 units of time, and by t_2 the action of including the generated object into the buffer. Notice that the initial tokens on p_5 represent the capacity of the buffer (3), and the arc connecting this place with t_2 is labelled by the interval $< 0, \infty >$, because these tokens can be consumed at any instant in the future. Tokens on p_6 represent the objects on the buffer which have not been yet consumed. Transition t_3 models the action of taking out an object from the buffer, which can occur at any instant. Finally, transition t_4 models the processing that makes the consumer for the objects extracted from the buffer, and this action takes between 4 and 6 units of time.

Let us observe that if the enabling time for the firing of one of these transitions (t_1 or t_4) expires, the system eventually becomes deadlocked, because we obtain a *dead* token either on p_1 or p_4.

Let us now see how we can fire transitions, and how we model the time elapsing.

Definition 2. (Firing rule)
Let $N = (P, T, F, times)$ be a TAPN, M a marking on it, and $t \in T$.

(i) We say that t is *enabled* at the marking M if and only if: $\forall p \in {}^\bullet t \ \exists x_p \in \mathbb{R}_0^+$ such that $M(p)(x_p) > 0 \wedge x_p \in times(p,t)$, i.e., on each precondition of t we have some token whose age belongs to $times(p,t)$.

(ii) If t is enabled at M, it can be fired, and by its firing we reach a marking M', defined as follows:

$$M'(p) = M(p) - C^-(p,t) + C^+(t,p), \ \forall p \in P$$

where both the subtraction and the addition operators work on multisets, and:

- $C^-(p,t) = \begin{cases} \{x_p\} & \text{if } p \in {}^\bullet t, x_p \in times(p,t) \\ & \text{and } x_p \in M(p) \\ \emptyset & \text{otherwise} \end{cases}$
- $C^+(t,p) = \begin{cases} \emptyset & \text{if } p \notin t^\bullet \\ \{0\} & \text{otherwise} \end{cases}$

Thus, from each precondition place of t we remove a token fulfilling (i), and we add a new token (with age 0) on each postcondition place of t.

As usual, we denote these evolutions by $M[t\rangle M'$, but it is noteworthy that these evolutions are in general non-deterministic, because when we fire a transition t, some of its precondition places could hold several tokens with different ages that could be used to fire it. Besides, we see that the firing of transitions does not consume any time. Therefore, in order to model the time elapsing we need the function *age*, defined below. By applying it we age all the tokens of the net by the same time:

(iii) The function $age : \mathcal{M}(N) \times \mathbb{R}_0^+ \longrightarrow \mathcal{M}(N)$ is defined by:

$$age(M,x)(p)(y) = \begin{cases} M(p)(y-x) & \textit{if } \ y \geq x \\ 0 & \textit{otherwise} \end{cases}$$

The marking obtained from M after x units of time without firing any transitions will be that given by $age(M,x)$. □

Although we have defined the evolution by firing single transitions, this can be easily extended to the firing of *steps* or *bags* of transitions; those transitions that could be fired together in a single step could be also fired in sequence in any order, since no *aging* is produced by the firing of transitions. In this way we

obtain step transitions that we denote by $M[R\rangle M'$. Finally, by alternating step transitions and time elapsing we can define a timed step semantics, where timed step sequences are those sequences $\sigma = M_0[R_1\rangle_{x_1} M_1 \ldots M_{n-1}[R_n\rangle_{x_n} M_n$, where M_i are markings, R_i multisets of transitions and $x_i \in \mathbb{R}_0^+$, in such a way that $M_i[R_{i+1}\rangle M'_{i+1}$ and $M_{i+1} = age(M'_{i+1}, x_{i+1})$. Note that we allow $x_i = 0$ in order to capture the execution in time zero of two causally related steps.

Then, given a MTAPN (N, M_0), we define $[M_0\rangle$ as the set of reachable markings on N starting from M_0, and we say that N is bounded if for every $p \in P$ there exists $n \in \mathbb{N}$ such that for all $M \in [M_0\rangle$ we have $|M(p)| \leq n$.

A token on a place p at a marking M is said to be *dead* if it can never be used to fire any transitions, i.e., it will remain on its place forever, just growing up. Thus, we say that a marking is *dead* when all its tokens are *dead*.

In a previous paper [13] we have shown that TAPNs have a greater expressiveness than PNs, even although TAPNs are not Turing complete, because they cannot correctly simulate a 2-counter machine. In that paper we proved that reachability is undecidable for TAPNs. Other properties that we have studied in a more recent paper [7] are coverability, boundedness and detection of dead tokens, which are all decidable for TAPNs. Decidability of coverability has been also proved in [1] for an extended version of TAPNs, in which all arcs can be annotated with bags of intervals in $\mathbb{N} \times (\mathbb{N} \cup \{\infty\})$.

2.2 Safe Timed-Arc Petri Nets

We say that (N, M_0) is safe if $|M(p)| \leq 1,\ \forall p \in P$, for every $M \in [M_0\rangle$. But let us observe that once a token becomes dead, it cannot be used for the firing of any transition in the future, and thus, we can extend the notion of safeness of a MTAPN taking into account that these tokens will not affect the behaviour of the net. Then, we may define a *soft-safe Timed-Arc Petri Net* as a MTAPN for which there is at most one non-dead token on each place for every reachable marking.

It is quite obvious that if N is a safe MTAPN, then it is also soft-safe, but the converse is not true in general. Furthermore, given a MTAPN N, if the underlying untimed net, $untimed(N)$ is safe, then N is also safe. However, the safeness of a MTAPN N does not imply the safeness of $untimed(N)$ (see Fig. 2).

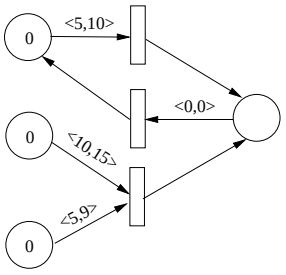


Fig. 2. A safe MTAPN N for which $untimed(N)$ is not safe

Moreover, as we can see in Fig. 3 a MTAPN N can be soft-safe, being unsafe its underlying untimed net.

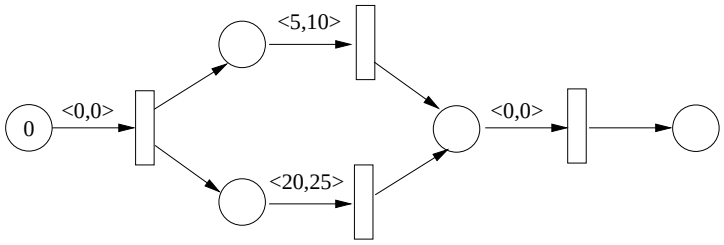


Fig. 3. A soft-safe MTAPN N for which $untimed(N)$ is not safe

3 State Graph

Let us observe that even in the case of bounded nets the number of reachable markings is infinite, because of the growing of the token ages. If we restrict ourselves to bounded timed-arc Petri nets with time intervals in $\mathbb{N} \times \mathbb{N} \cup \{\infty\}$, we may follow the same ideas in [8], in order to construct a state graph, by using the fact that for every place p of a timed-arc Petri net we can find a maximal value $Max(p)$ for the age of its tokens to influence the activation of its postcondition transitions, because the tokens on that place with an age exceeding that maximal value can only be consumed by the firing of some transition $t \in p^\bullet$ for which $\pi_2(p,t) = \infty$.

Concretely we may define:

$$Max(p) = Max\{\pi_i(p,t) \mid t \in p^\bullet,\ \pi_i(p,t) < \infty,\ i = 1,2\}$$

and $S(p) = 1 + Max(p)$. Then, it follows that once the age of a token on p exceeds $Max(p)$ the only postcondition transitions $t \in p^\bullet$ that could be fired by using that token are those for which $\pi_2(p,t) = \infty$. Obviously, in order to fire such a transition t the age of the involved token on p is unimportant once it exceeds $S(p)$. This means that in order to construct the state graph we can represent by the single value $S(p)$ the whole interval $[S(p), \infty]$.

Definition 3. (State graph)
Given a bounded MTAPN $N = (P, T, F, times, M_0)$ with time intervals in $\mathbb{N} \times (\mathbb{N} \cup \{\infty\})$, we define its state graph $G(N) = (V, E, M_0)$, where V is the set of markings of N such that the age of the tokens of every place $p \in P$ belongs to $\{0, \ldots, S(p)\}$, M_0 is the initial state and $E \subset V \times T \times \mathbb{N} \times V$.

An arc $(M, t, r, M') \in E$ means that t can be fired at the marking $age(M, r)$ applying Def. 2, and M' is the reached state, but changing for every place p the age of the tokens exceeding $Max(p)$ to $S(p)$.

Thus, we start from M_0 and we apply the modified firing rule to successively get the new reachable states of the graph. We must observe that as we are not imposing the urgent firing of transitions, in principle we could have infinitely many arcs leaving each state. However, for each state $M \in V$ we have that from a certain instant $r \in \mathbb{N}$ onwards, all its outgoing arcs will reach the same state, because the ages of the tokens in each place $p \in P$ at the markings $age(M, s)$, for $s \geq r$, would be greater than their corresponding maximal value, $Max(p)$. As a consequence, for each state $M \in V$ we can limit the application of the firing rule to the markings obtained aging M up to time $r_{max} = Max\{S(p) \mid p \in P\}$.

According to this definition, taking into account that V is finite and that we have a finite number of arcs leaving each state, we conclude that $G(N)$ is finite.

□

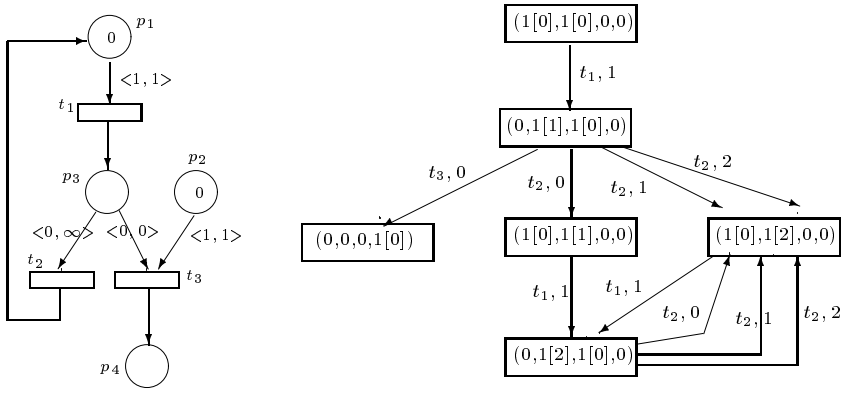


Fig. 4. A bounded MTAPN and the corresponding state graph

In Fig. 4 we can see a bounded MTAPN and its corresponding state graph. For this net we have that $S(p_1) = 2, S(p_2) = 2, S(p_3) = 1, S(p_4) = 0$ and thus $r_{max} = 2$. Graph states are shown by indicating for each place the number of tokens of a certain age that we have on this place, for instance $1[0]$ means that we have one token with age 0 in the corresponding place.

Then, we have the following result which relates reachable markings of N with states in $G(N)$.

Proposition 1. Given a bounded MTAPN $N = (P, T, F, times, M_0)$ with time intervals in $\mathbb{N} \times (\mathbb{N} \cup \{\infty\})$, and its corresponding state graph $G(N) = (V, E, M_0)$. If $M \in [M_0\rangle$, then there is a state $M' \in V$ such that $\varphi(M) = \varphi(age(M', r))$, for a certain $r \in \mathbb{N}$, where φ is a function that for every place p changes the age of the tokens on p exceeding $Max(p)$ to $S(p)$:

$$\varphi : \mathcal{M}(N) \longrightarrow V$$

$$\varphi(M)(p)(n) = \begin{cases} M(p)(n) & \text{if } n \leq Max(p) \\ \sum_{m \geq S(p)} M(p)(m) & \text{if } n = S(p) \\ 0 & \text{if } n > S(p) \end{cases}$$

Proof. By induction on the timed step sequence σ such that $M_0[\sigma\rangle M$. The base case ($M = M_0$) is trivial, so let us consider a reachable marking M_1 for which we have a state $M_1' \in V$ such that $\varphi(M_1) = \varphi(age(M_1', r_1))$, and both possibilities of evolution, either by the simple passage of time or by firing a transition:

- If $M = age(M_1, r_2)$, we have that $\varphi(M) = \varphi(age(M_1', r_1 + r_2))$, as desired.
- If $M_1[t\rangle M$, we have that t is also enabled at $\varphi(age(M_1', r_1))$, because the change of age that φ makes for the tokens exceeding their maximal value does not affect the enabledness of transitions. Besides, we may use for firing t the adequate tokens to get a state M' such that $\varphi(M) = \varphi(M') = \varphi(age(M', 0))$.

□

Thus, the state graph provides us with a necessary condition for a marking to be reachable in a bounded MTAPN, but in general it will not allow us to determine if a given marking is reachable or not.

In the particular case of safe MTAPNs, the state graph will be a little smaller, because we will have at most one single token on each place for every reachable marking. For soft-safe MTAPNs the previous construction will not generate in general a finite graph (if the net is unbounded). However, dead tokens do not affect the future behaviour of the net (in terms of the transitions that we may execute), and thus, we may adapt the previous construction for the particular case of soft-safe MTAPNs.

Definition 4. (State graph for soft-safe MTAPNs)
Given a MTAPN $N = (P, T, F, times, M_0)$ with time intervals in $\mathbb{N} \times (\mathbb{N} \cup \{\infty\})$, we may define its state graph $G^*(N) = (V, E, M_0)$, in a similar way to that followed in Def. 3, but in this case, once we have obtained an arc (M, t, r, M') according to Def. 3, we check for every place $p \in P$ if there are two or more tokens with the same age on p at M', and in this case we only consider one of them, i.e., we will finally take an arc (M, t, r, M''), where:

$$M''(p)(j) = Min\{1,\, M'(p)(j)\} \;\; \forall j \in \{0, \ldots, S(p)\}$$

With this construction we can generate only a finite number of possible states, and thus the graph is finite. □

With this new construction we have removed some tokens which appear across the normal evolution of the net, but the important fact is that these tokens are *innocuous*, in the sense that they do not affect the future behaviour of the net, because we are supposing that it is soft-safe. Of course, we must pay an additional price for that: we are losing some more information concerning reachability.

Proposition 2. Given a soft-safe MTAPN $N = (P, T, F, times, M_0)$ with time intervals in $\mathbb{N} \times (\mathbb{N} \cup \{\infty\})$, and its corresponding state graph $G^*(N) = (V, E, M_0)$. If $M \in [M_0\rangle$, then there is a state $M^* \in V$ such that $\varphi(M) \geq \varphi(age(M^*, r))$, for a certain $r \in \mathbb{N}$, where:

- $M_1 \geq M_2$ if and only if $\forall p \in P : M_1(p)(n) \geq M_2(p)(n)$, $\forall n \in \{0, \ldots, S(p)\}$.
- φ is defined as in Prop. 1.

Moreover, for every place $p \in P$, if there is one non-dead token in p at M, then we have one token in $age(M^*, r)$ with either the same age or both have an age greater or equal than $S(p)$.

Proof. We use again induction on the timed step sequence σ such that $M_0[\sigma\rangle M$. The base case ($M = M_0$) is trivial, so let us consider a reachable marking M_1 for which we have a state $M_1^* \in V$ fulfilling:

- $\varphi(M_1) \geq \varphi(age(M_1^*, r_1))$.
- Every non-dead token at M_1 has its counterpart in $age(M_1^*, r_1)$.

Then, let us consider both possibilities of evolution, either by time elapsing or by firing one transition:

- If $M = age(M_1, r_2)$, it follows that $\varphi(M) = \varphi(age(M_1, r_2)) \geq \varphi(age(M_1^*, r_1 + r_2))$, and every non-dead token in M was non-dead in M_1 too, thus we can apply the induction hypothesis for it, obtaining its counterpart in $age(M_1^*, r_1)$. We just need to age this token in r_2 units of time in order to obtain the token in $age(M_1^*, r_1 + r_2)$.
- If $M_1[t\rangle M$, it follows that t is also enabled at $\varphi(age(M_1^*, r_1))$, because every non-dead token of M_1 has its counterpart in $age(M_1^*, r_1)$ and N is soft-safe. Let M^* be the state that we obtain by applying Def. 4, it can be easily checked that:
  - $\varphi(M) \geq \varphi(M^*) = \varphi(age(M^*, 0))$.
  - If we generate a new non-dead token on a place $p \in P$, this token will have its counterpart in M^*, because the firing of t would generate a new token of age 0 there, unless we already had one token of age 0 in p.

□

Therefore, with this proposition we have again a necessary condition for a marking to be reachable in a soft-safe MTAPN. Notice that safeness of a MTAPN can be effectively decided by using an enumerative algorithm, taking into account the normalization introduced in Def. 3. Specifically, we generate the state graph until reaching a node in which a place has two tokens, in this case the net will not be safe, and the algorithm finishes. If the construction terminates with a state graph in which all the states have a single token on each place, we will have a safe net. This algorithm can be slightly modified to effectively decide whether a MTAPN is soft-safe or not. We annotate a maximum of two tokens with the same age on each place when constructing the graph, and each token is labelled with a *boolean value*, indicating if the token can be consumed in the future. When

the token is generated this value is *false*, but if we detect along the generation that one token can be used to fire one transition, we relabel the token with the value *true* all along the graph. If the net is soft-safe we will never generate two tokens in the same place with values *true* for a same state, and the algorithm will terminate because the number of possible states is finite. On the other hand, when the net is not soft-safe, there must be a sequence of transitions reaching a state in which two tokens staying in a place can be eventually consumed, and thus, we will terminate the execution saying that the net is not soft-safe.

4 MPEG-2 Encoding Algorithm

MPEG standards were designed with two requirements in mind, namely, the need for a high compression, and the need for random access capability. These techniques exploit the fact that video sequences usually contain statistical redundancies in both temporal and spatial directions. Thus, MPEG digital video coding techniques are statistical in nature. Specifically, the basic statistical property upon which MPEG compression techniques rely is inter–pixel region correlation. The contents of a particular pixel region can be predicted from nearby pixel regions within the same frame (intra–frame coding) or from pixel regions of a nearby frame (inter–frame coding).

Perhaps the ideal method for reducing temporal redundancies is one that tracks every pixel from frame to frame. However, this extensive search is computationally expensive. Under the MPEG standards, this search is performed by tracking the information within 16×16 pixels regions, called *macroblocks*. Given two contiguous frames, $frame(t)$ and $frame(t-1)$, for each macroblock in $frame(t)$, the encoder determines the best matching macroblock in $frame(t-1)$ and calculates the *motion vector*, which captures the macroblock translation information. Therefore, the *temporal redundancy reduction processor* generates a representation for $frame(t)$ by using the corresponding macroblock from $frame(t-1)$, and this representation only contains the motion vector and the prediction error (changes between both frames). This technique is called *motion compensated prediction*.

In order to reduce spatial redundancies a DCT (Discrete Cosine Transform) is used. With this coding process some subjective redundancies in the image are removed, on the basis of human visual criteria.

The combination of these two techniques described above are the key elements of the MPEG encoding process. Furthermore, in order to achieve the requirement of random access and high compression, the MPEG–2 standard specifies three types of compressed video frames: I pictures, P pictures and B pictures. I pictures (intracoded pictures) are coded with no reference to other frames, exploiting only spatial correlation in a frame. P pictures (predictive coded pictures) are coded by using motion compensated prediction of a previous I or P picture. Finally, B pictures (bidirectionally–predictive coded pictures) are obtained by motion compensation by using past and future reference frames (I or P pictures).

A group of consecutive I, P and B pictures constitute a structure called Group of Pictures (GoP). Therefore, a video sequence may be seen as a sequence of GoPs.

The block diagram of the MPEG encoder is depicted in Figure 5. In order to understand how the MPEG–2 encoder works, we will consider a typical GoP consisting on the frames IBBP. Despite B pictures appear before P pictures, the encoding order is IPBB because B pictures require both past and future frames as references.

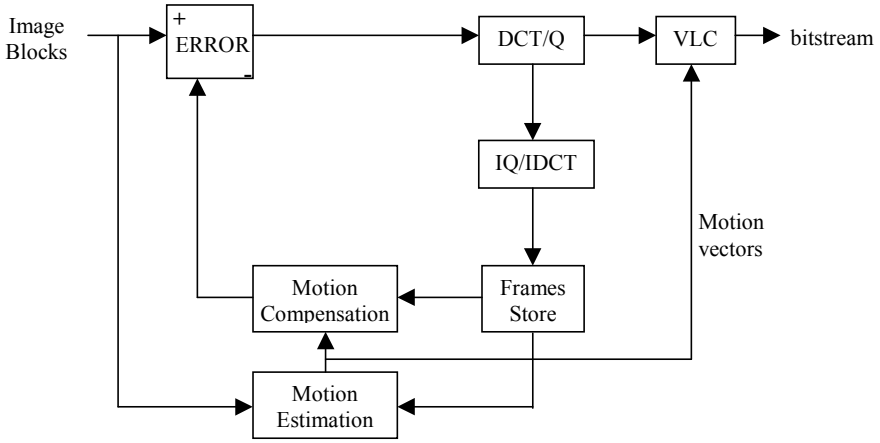


Fig. 5. Block diagram of the MPEG–2

The first frame in a GoP (I picture) is encoded in intra mode without references to any past or future frames. The DCT is applied to each macroblock and then it is uniformly quantized (Q). After quantization, it is encoded using a variable length code (VLC) and it is sent to the output buffer. At the same time the reconstruction (IQ) of all non–zero DCT coefficients belonging to one macroblock and the Inverse DCT (IDCT) give us a compressed I picture which is stored temporarily in the *Frame Store (FS)*.

When the input is coded either as P or B pictures, the encoder does not code the picture macroblocks directly. Instead, it codes the prediction errors and the motion vectors. With P pictures, for each macroblock in the current picture, the motion estimation gives us the coordinates of the macroblock in the I picture that best matches its characteristics and thus, the motion vector may be calculated. The motion compensated prediction error is obtained by subtracting each pixel in a macroblock with its motion shifted counterpart in the previous frame. The prediction error and the motion vectors are coded (VLC) and sent to the output buffer. As in the previuos case, a compressed P picture is stored in the Frame Store.

With B pictures, the motion estimation process is performed twice: for a past picture (I picture in this case), and for a future picture (P picture). Prediction errors and both motion vectors for each macroblock are coded (VLC) and sent to the output buffer. Notice that the compressed B pictures are not stored in the Frame Store, since they are not needed to calculate any other pictures.

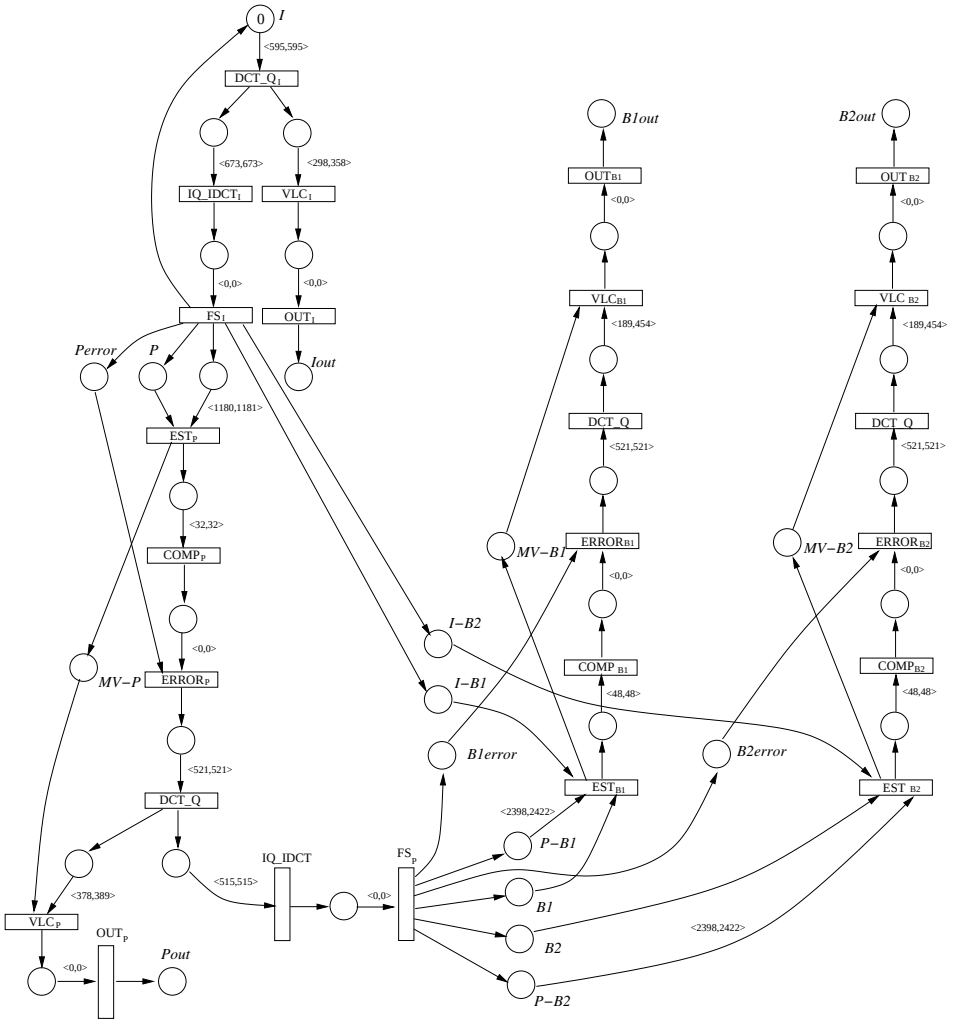


Fig. 6. TAPN which models the MPEG–2 codification process of a GoP

4.1 Timed-Arc Petri Net modeling the MPEG-2

Figure 6 shows the timed-arc Petri net modeling the MPEG-2 encoding algorithm[3]. The left-hand side describes the first part of the encoding algorithm, which corresponds to the generation of both I and P encoded pictures (but remember that even if P pictures are generated before the B pictures, they will appear the last in the final video sequence). Once the places *I-B1, I-B2, P-B1, P-B2* are marked the second part of the net becomes activated (right-hand side),

[3] For simplicity, we omit in this picture the label of the arcs when they are labelled by $<0, \infty>$.

which models the *B1* and *B2* picture encoding process. *Iout* (*Pout* resp.) represents the output of an encoded *I-picture* (*P-picture*), while places *B1out* and *B2out* represent the output of B1 and B2 pictures.

The time intervals that label the arcs connecting places with transitions have been obtained from several real measurements, by coding the "Composed" Video sequence. This experiment has been repeated a number of times, and the results being reported below are therefore the minimum and the maximum of all these trials[4]. During these trials, no other operations were taken place in our experimental setup. The "Composed" Video sequence (format PAL CCIR601, 720x576 pixels) is a representative video sequence which has several different motion levels, and we have encoded it by using a completely software-based MPEG-2 video encoder derived from that developed in Berkeley, which is freely available in the MPEG Home Page: *http://drogo.cslet.stet.it/mpeg*.

In order to get the real values for the different elements of the encoder we have included some patches into the source code, which correspond with the beginning and the end of the elementary actions that we have described in the specification of the algorithm. The real values thus obtained for I and P pictures being in the output buffer are shown in Table 1, as well as the times for encoding the complete GoP.

Picture	Min	Max
I	893 ms	953 ms
P	3688 ms	3738 ms
GoP	11567 ms	12085 ms

Table 1. Measured real values

5 Analysis and Conclusions

We may compute the times to reach every place of a TAPN, by constructing a state reachability graph. With that process we obtain a time interval for every place. In our case, for the TAPN modeling the MPEG–2 encoder we have obtained the results shown in Table 2.

Picture	Min 1st GoP	Max 1st GoP	Min Nth GoP	Max Nth GoP
I	893 ms	953 ms	893 + 1268 * (N-1) ms	953 + 1268 * (N-1) ms
P	3379 ms	3391 ms	3379 + 1268 * (N-1) ms	3391 + 1268 * (N-1) ms
GoP	6672 ms	6962 ms	6672 + 1268 * (N-1) ms	6962 + 1268 * (N-1) ms

Table 2. Values obtained from the TAPN

[4] These values were obtained in a single processor Pentium II - 350MHz platform with 64MB RAM.

Comparing these results with those shown in Table 1 we can see that there are strong differences in order to complete the encoding process of a GoP. For instance, to complete the first GoP the measured values are $[11567\ ms, 12085\ ms]$ while in the TAPN we have obtained $[6672\ ms, 6962\ ms]$. These strong differences are due to the important fact that the encoder only uses a single processor, whereas our TAPN model captures all the intrinsic parallelism of the encoding process, so that with the analysis of the TAPN we are obtaining the required times provided that we have as many processors as needed to take advantage of this parallelism (for a single GoP two processors would be enough).

Let us now consider the encoding process of N GoPs, according to the TAPN the encoding would take $[6672 + 1268 * (N-1)\ ms, 6962 + 1268 * (N-1)\ ms]$, and thus, the average time for encoding each GoP would converge to 1268 ms. This can be interpreted as the time needed to encode a GoP for an infinite video sequence provided that the number of processors is big enough.

Fig. 7 shows the number of processors required in order to process any sequence of GoPs, according to the TAPN of Fig. 6. From that Figure we may conclude that the best performance could be obtained with **10 processors**.

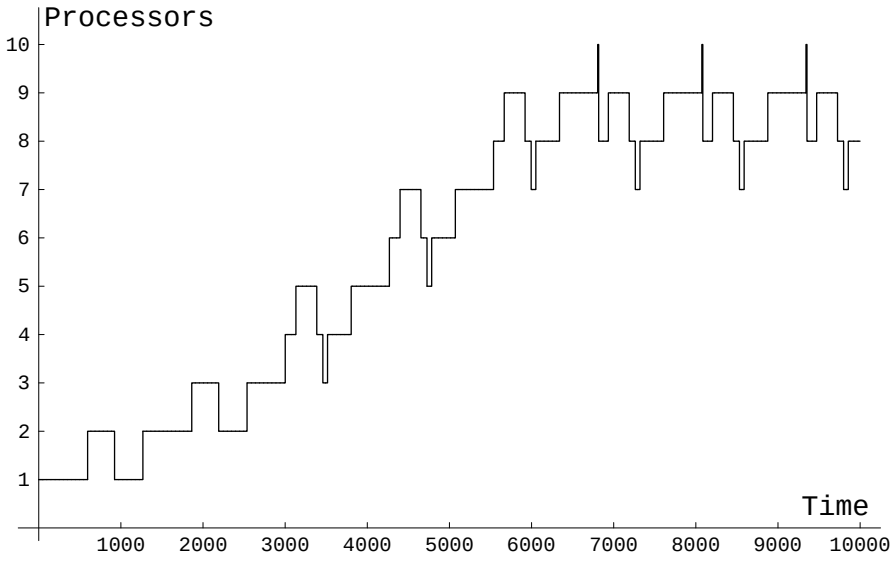


Fig. 7. Graph relating the number of required processors and time (ms)

Consequently, the main conclusion from this analysis of the MPEG–2 encoder by using TAPNs is that the performance of the encoding algorithm for a single GoP can be improved in a factor near to 50% by using two processors. Moreover, in longer sequences of GoPs this factor can be increased, and we have obtained an average time of 1268 ms for encoding each single GoP of an infinite video sequence, provided that we have as many processors as required (10 in this particular case).

References

1. P.A. Abdulla and A. Nylén. *Timed Petri Nets and BQOs.* In Proc. ICATPN 2001, Lecture Notes in Computer Science, vol. 2075, pp. 53-70, 2001.
2. W.M.P. van der Aalst. *Interval Timed Coloured Petri Nets and their Analysis.* Lecture Notes in Computer Science, vol. 691, pp. 451-472. 1993.
3. M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte and A. Cumani. *On Petri Nets with Stochastic Timing.* Proc. of the International Workshop on Timed Petri Nets, IEEE Computer Society Press, pp. 80-87. 1985.
4. T. Bolognesi, F. Lucidi and S. Trigila. *From Timed Petri Nets to Timed LOTOS.* Proc. of the Tenth International IFIP WG6.1 Symp. on Protocol Specification, Testing and Verification. 1990.
5. Fred D.J. Bowden. *Modelling time in Petri nets.* Proc. Second Australia-Japan Workshop on Stochastic Models. 1996.
6. Antonio Cerone and Andrea Maggiolo-Schettini. *Time-based expressivity of time Petri nets for system specification.* Theoretical Computer Science (216)1-2, pp. 1-53. 1999.
7. D. de Frutos, V. Valero and O. Marroquín. *Decidability of Properties of Timed-Arc Petri Nets.* Proc. ICATPN 2000, Lecture Notes in Computer Science, vol. 1825, pp. 187-206. 2000.
8. Hans-Michael Hanisch. *Analysis of Place/Transition Nets with Timed-Arcs and its Application to Batch Process Control.* Application and Theory of Petri Nets, LNCS vol. 691, pp:282-299. 1993.
9. ISO/IEC 13818-2. *Draft International Standard Generic Coding of Moving Pictures and Associated Audio.* Recommendation H.262.
10. P. Merlin. *A Study of the Recoverability of Communication Protocols.* PhD. Thesis, Univ. of California. 1974.
11. C. Ramchandani. *Performance Evaluation of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD. Thesis, Massachusetts Institute of Technology, Cambridge. 1973.
12. J. Sifakis. *Use of Petri Nets for Performance Evaluation.* Proc. of the Third International Symposium IFIP W.G.7.3., Measuring, Modelling and Evaluating Computer Systems. Elsevier Science Publishers, pp. 75-93. 1977.
13. V. Valero, D. de Frutos and F. Cuartero. *On Non-decidability of Reachability for Timed-Arc Petri Nets.* Proc. 8th Workshop on Petri Nets and Performance Models, PNPM'99, pp. 188-196. 1999.
14. B. Walter. *Timed Petri-Nets for Modelling and Analysing Protocols with Real-Time Characteristics.* Proc. 3rd IFIP Workshop on Protocol Specification, Testing and Verification, North-Holland. 1983.

PAMR: A Process Algebra for the Management of Resources in Concurrent Systems*

Manuel Núñez and Ismael Rodríguez

Dept. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, E-28040 Madrid. Spain.
{mn,isrodrig}@sip.ucm.es

Abstract In this paper we present a *process algebra for the management of resources* in concurrent systems. Our aim is to define a formal framework that can help in the task of specifying systems that depend, for their execution, on a set of *resources* that they use. Usually, systems consist in a set of processes. In order to improve their performance, these processes will be able to exchange resources among them. In our language, processes will consist in a *behavior* (formalized as a LOTOS process) and in information about the resources that they own. Systems will be defined as the parallel composition of a set of processes. We will study some examples applying the features of PAMR. These examples will try to show the usefulness of our language for specifying and analyzing concurrent systems where resources play an important role.

1 INTRODUCTION

During the last two decades Process Algebras have been used to specify and verify different kinds of systems. Nevertheless, most process algebraic models lack the ability to appropriately express the relation between the behavior of processes and the resources that they use. Actually, resources are usually modeled as *processes*. In this paper we will present a process algebra to deal with systems where resources must be taken into consideration. In order to illustrate the kind of systems that we will deal with, let us introduce the following simple running example. Let us consider a system consisting in the parallel execution of n subsystems ($P_1 \ldots P_n$) and m different kinds of resources that these subsystems may use (let us suppose that the total quantity of the resource i is equal to x_i). The performance of these subsystems depends on these resources (for example, the portion of memory used by each subsystem, time quantum of CPU, time quantum of access to the bus, etc). Each subsystem P_j has an initial distribution of resources $x_1^j \ldots x_m^j$, that is, in the beginning, subsystem j owns x_i^j units of the resource i. Given the fact that the quantity of resources that subsystems own cannot be bigger than the total amount, we work under the constraint:

* Research supported in part by the CICYT project TIC 2000-0701-C02-01.

$\forall\ 1 \leq i \leq m :\ \sum_j x_i^j \leq x_i$. Finally, let us suppose that subsystems have a *preference* on how they *like* resources. For example, suppose that a subsystem P_{j_1} runs at the same speed if we replace one unit of the resource i_1 by four units of the resource i_2, while another subsystem P_{j_2} runs at the same speed if one unit of the resource i_1 is replaced by two units of the resource i_2. In particular, P_{j_1} will perform better if we replace three of its units of i_2 by one additional unit of i_1.

In order to describe this kind of systems, we will introduce a language where the usual description of the behavior of processes is extended with additional information. In our language, a process will not only take into account its activity (that is, which actions can be performed at a given point of time) but it will also consider which resources can be used during its execution. Besides, subsystems may *exchange* resources with other subsystems. For instance, in the example given above, if P_{j_1} gives to P_{j_2} three units of i_2 and receives from P_{j_2} one unit of i_1, then both subsystems run faster. So, our language will provide mechanisms to, starting with an initial distribution of resources, accomplish a better performance of the studied system.

Example 1. Consider a system where two programs are running. Using operating systems terminology suppose that one of the programs is an I/O-bound process (e.g. a process printing a paper) while the other one is a CPU-bound process (e.g. a process compiling the LaTeX source of another paper). Even if the initial distribution would assign the same resources to both processes (in terms of CPU use and access to I/O), it is clear that both processes would perform better if they exchange their resources in the adequate way. That is, even though the I/O-bound process will keep some time quantum of CPU, it is willing to reduce it (with respect to the original distribution) in order to get better access to I/O, and the other way around for the CPU-bound process. □

We will define our language in two steps. First, we consider a *base language* where we can specify the usual behavior of *processes*. For this purpose, we will use a LOTOS like language. As we already said, processes will also contain information about the available resources: The resources that they own, the resources that they need to perform actions, etc. In particular, there will be a function indicating their preferences on resources: A *utility function*. This function will compute values according to the actions that the process is able to perform immediately. For example, consider a program that initially behaves as a CPU-bound process and after a while its behavior is as an I/O-bound process. This must be reflected in its utility function. In the beginning, it will prefer to keep a bigger time quantum of CPU, but in the end, its utility function will report bigger values if it keeps better access to I/O.

The second step consists in the combination of processes to build *systems*. A system will be the parallel composition of a set of processes. Processes will be able to communicate with each other by two kinds of operations. First, we will have the usual mechanism for synchronization in parallel operators, that is, there will be some actions that processes may perform asynchronously and

some actions that they have to perform synchronously. More important, in order to improve their performances, processes will be able to exchange resources. Besides, *harmful* exchanges will be forbidden. For instance, it will not be allowed that P_{j_2} exchanges three units of i_2 by one unit of i_1 with P_{j_1} because both subsystems get worse, and thus the whole system deteriorates. We will consider a parallel operator where communications are not restricted to be exactly between two processes. For instance, we will allow that three processes synchronize in an action a or that four processes exchange, in one step, resources among them.

Regarding exchange of resources, we will consider two *policies*. The first one, that we call *preserving utility*, will allow exchanges of resources only in the case that at least one of the processes improves its situation, and no process gets worse after the exchange. For example, this policy would not allow P_{j_1} to *trade* with P_{j_2} one unit of i_1 by one unit of i_2, because even though P_{j_1} improves its situation we have that P_{j_2} gets worse. The second policy, that we call *maximizing utility*, will not be based on the particular situations of processes but in the situation of the system. In this case, exchanges will be allowed only if they improve the overall performance of the system. For example, this policy would allow the previous exchange only if the *profit* for P_{j_1} is bigger than the *loss* for P_{j_2}. In fact, this policy is very similar to consider that processes are not the owner of resources. This is so because part of the resources of a process can be *expropriated* (as far as the whole system improves).

In terms of related work, our language is based on a previous proposal [12]. In this paper we have extended that language to get a more appropriate framework for the specification of the systems that we are dealing with. Nevertheless, the theoretical framework introduced in [12], where some concepts of microeconomic theory were used, can be inherited by adding some slight modifications. There are models to specify systems sharing resources (e.g. [5]), but in this case resources are just accessed, not traded; this access induces some delays in the behavior of processes. If we consider the maximizing utility policy, our proposal is somehow related to the *ODP trading function* [9]. Nevertheless, under a preserving utility policy, it is the case that a process only uses (and nobody else can use them) the resources that it owns. Besides, under this policy, trade permanently transfers the *ownership* of the traded resources. Finally, management of resources appears in fields like operating systems or concurrent programming. Resources are usually owned by a *mediator* which allows the processes to use them. So, given the fact that we separate the behavior of a process and how it manages its resources, we think that our language can be successfully used for the specification of this kind of systems.

The paper is organized as follows. In Section 2 we define our language in terms of processes and systems. In Section 3 we present two examples of the use of our language and we study their properties. Finally, in Section 4 we present our conclusions and some directions for further research. An extended version of this paper, where a bisimulation semantics is defined and more complex examples are presented, can be found in [13].

2 DEFINITION OF THE LANGUAGE

In this section we present PAMR as well as its operational semantics. First, we will introduce the notion of *process*. Then, we will say that a system is the parallel composition of several *communicating* processes. We will define the operational semantics of systems by means of three rules. The first rule will describe how processes exchange resources among them. Processes will exchange resources until no more *useful* exchanges are possible. Then, the last two rules define how systems perform actions (possibly by synchronizing among the corresponding processes). We will finish this section by presenting some useful results for verifying some properties in Section 3.

Before we present the formal definition of our language, we introduce some mathematical notation which will be used in the rest of the paper.

Definition 1. We consider $\mathbb{R}_+ = \{x \in \mathbb{R} \mid x \geq 0\}$. By abuse of notation, we will consider $\frac{r}{0} = \infty$. Let $r \in \mathbb{R}_+$. Then, $\mathtt{trunc}(r)$ denotes the natural number resulting from discarding the decimal part of r (e.g. $\mathtt{trunc}(1.2) = 1$).

We will usually denote *vectors* in $\mathbb{R}^n$ (for $n \geq 2$) by $\bar{x}, \bar{y}, \ldots$. Given $\bar{x} \in \mathbb{R}^n$, x_i denotes its *i-th* component. We extend some usual arithmetic operations to vectors. Let $\bar{x}, \bar{y} \in \mathbb{R}^m$. We define $\bar{x} + \bar{y} = (x_1 + y_1, \ldots, x_n + y_n)$. We write $\bar{x} \leq \bar{y}$ if for any $1 \leq i \leq n$ we have $x_i \leq y_i$.

We will usually denote *matrices* in A^{n*m} (for $n, m \geq 2$, and a set A) by calligraphic letters $\mathcal{E}, \mathcal{E}_1 \ldots$. Let $\mathcal{A} \in A^{n*m}$, and i, j be such that $1 \leq i \leq n$ and $1 \leq j \leq m$. We will denote the $(n*(i-1)+j)th$ component of $\mathcal{A}$ by $\mathcal{A}_{ij}$, that is, if we consider $\mathcal{A}$ as a matrix, this component corresponds to the element located at file i and column j.

Given a set A, $\mathcal{P}(A)$ denotes the set containing all the subsets of A. □

The behaviors of processes will be defined by means of a usual process algebra. In this case, we will consider a simple LOTOS like *base language*. First, we give an auxiliary definition introducing the sorts for actions.

Definition 2. Let $\mathtt{Act}$ be the set of *visible* actions ($a, b \ldots$ range over $\mathtt{Act}$). Let $\mathtt{Act}_\tau$ be the set of *internal* actions ($\mu, \mu' \ldots$ range over $\mathtt{Act} \cup \mathtt{Act}_\tau$). We suppose that $\tau \in \mathtt{Act}_\tau$, that there exists a bijection $f : \mathtt{Act} \longrightarrow (\mathtt{Act}_\tau - \{\tau\})$, and $\mathtt{Act} \cap \mathtt{Act}_\tau = \emptyset$. Given a visible action $a \in \mathtt{Act}$, we will denote $f(a)$ by τ_a. We denote by $\mathtt{ACT}$ the *set of actions*, that is, $\mathtt{ACT} = \mathtt{Act} \cup \mathtt{Act}_\tau$. Finally, let Id be a set of (basic) process identifiers. □

Let us note that we will have not only a unique internal action, but a whole set of them. In addition to the usual τ action, we consider an internal action for each of the visible actions. For an *external* observer, all the internal actions will be equal. The difference among them comes from the fact that they will need different resources to be performed. So, if a process needs a set of resources $\bar{x}$ to perform a visible action a, and this action is *hidden*, the resulting action, that is τ_a, will need the same amount of resources $\bar{x}$ to be performed. Sets of internal actions appear in other models for concurrent processes (for example, for I/O automata [10]).

$$(\text{PRE})\ \frac{}{\mu;B \xrightarrow{\mu} B} \qquad (\text{PAR1})\ \frac{B_1 \xrightarrow{\mu} B_1' \wedge \mu \notin A}{B_1 \|_A B_2 \xrightarrow{\mu} B_1' \|_A B_2}$$

$$(\text{CHO1})\ \frac{B_1 \xrightarrow{\mu} B_1'}{B_1 + B_2 \xrightarrow{\mu} B_1'} \qquad (\text{PAR2})\ \frac{B_1 \xrightarrow{a} B_1' \wedge B_2 \xrightarrow{a} B_2' \wedge a \in A}{B_1 \|_A B_2 \xrightarrow{a} B_1' \|_A B_2'}$$

$$(\text{CHO2})\ \frac{B_2 \xrightarrow{\mu} B_2'}{B_1 + B_2 \xrightarrow{\mu} B_2'} \qquad (\text{PAR3})\ \frac{B_2 \xrightarrow{\mu} B_2' \wedge \mu \notin A}{B_1 \|_A B_2 \xrightarrow{\mu} B_1 \|_A B_2'}$$

$$(\text{HID1})\ \frac{B_1 \xrightarrow{\mu} B_1' \wedge \mu \notin A}{\text{hide } A \text{ in } B_1 \xrightarrow{\mu} \text{hide } A \text{ in } B_1'} \qquad (\text{HID2})\ \frac{B_1 \xrightarrow{a} B_1' \wedge a \in A}{\text{hide } A \text{ in } B_1 \xrightarrow{\tau_a} \text{hide } A \text{ in } B_1'}$$

$$(\text{REC})\ \frac{B\{X:=B/X\} \xrightarrow{\mu} B'}{X:=B \xrightarrow{\mu} B'}$$

Figure1. Operational Semantics for the Base Language.

Definition 3. The set of *basic processes*, denoted by $\mathcal{B}$, is given by the BNF-expression $B ::= \text{stop} \mid X \mid \mu\,;B \mid B + B \mid B \|_A B \mid \text{hide } A \text{ in } B \mid X := B$, where $\mu \in \texttt{ACT}$, $A \subseteq \texttt{Act}$, and $X \in Id$. □

The operational semantics for the base language is given in Figure 1, and it is standard. Let us remind that $B\{B'/X\}$ represents the replacement of the free occurrences of the variable X in B by the term B'. Let us also remark that, in rule (HID2), the result of hiding a visible action a is not τ but τ_a. The following definition will be used later on. It computes the actions that a basic process may perform immediately.

Definition 4. Let $B \in \mathcal{B}$. We define its *set of immediate actions*, denoted by $\texttt{Imm}(B)$, as the set $\texttt{Imm}(B) = \{\mu \in \texttt{ACT} \mid \exists\, B' \in \mathcal{B} :\ B \xrightarrow{\mu} B'\}$. □

As we sketched in the introduction, a process will not only consist in a behavior. On the contrary, a process will keep track of the resources assigned to it and of some information relating resources and actions. Specifically, a process P will be defined as a tuple $(B, \bar{x}, u, n, c)$ where the intuitive meaning of the components is:

- $B \in \mathcal{B}$ is a *basic process* indicating the behavior of P.
- $\bar{x} \in \mathbb{R}_+^m$ indicates that P owns x_i units of the *i-th* resource, for all $1 \leq i \leq m$.
- u is a function indicating *preferences* between *baskets* of resources. This function takes as parameters a set of actions (the actions that B may perform immediately) and a set of resources, and returns a real number. For example, $u(A, \bar{x}) < u(A, \bar{y})$ means that if $\texttt{Imm}(B) = A$ then P would prefer to own the basket $\bar{y}$ to $\bar{x}$. So, processes will try to increase the value of u, given A, by exchanging resources with other processes.
- n is a function relating *resources* and speed of execution. This will be modeled by a function taking as parameters an action and a set of resources. In particular, $n(a, \bar{x}) = \infty$ means that a needs more than $\bar{x}$ to be performed;

$n(a, \bar{x}) = r \in \mathbb{R}_+$ means that a takes r units of time to be performed if the process has the use of the resources $\bar{x}$. A necessary condition for P to perform a is $n(a, \bar{x}) < \infty$. Let us remark that in some situations there will be a strong relation between the functions u and n: Bigger utilities will be produced by bigger amounts of (or better) resources, and this will imply faster performance of actions. So, sometimes we may have relations like $n(a, \bar{x}) = \frac{K}{u(\{a\}, \bar{x})}$, for a given constant $K \in \mathbb{R}_+$. Also note that this is not always the situation. For instance, there can be actions that do not need any resources to be performed, actions that need only a strict subset of them, etc. That is why we have preferred to keep both functions.

- c is a function indicating the *consumed* resources after performing an action. In some situations, the execution of an action needs to consume some resources (for example, writing in a bounded buffer); in other situations, resources are created after performing actions (for example, a pop operation from a bounded stack). These situations will be modeled by a function that takes as inputs an action and a set of resources, and returns a set of resources. That is, $c(a, \bar{x}) = \bar{y}$ means that, after performing a, the set of resources of the process varies from $\bar{x}$ to $\bar{y}$. For example, if $\bar{z} = c(a, \bar{x}) - \bar{x}$ then $z_i > 0$ indicates that, after performing a, the process has created z_i units of the *i-th* resource. A necessary condition for a process to perform an action a is $c(a, \bar{x}) \geq \bar{0}$. We do not allow *debts* because they could generate inconsistencies. For example, such debts could produce that two processes use simultaneously a printer.

Definition 5. Let us consider that there exists a number $m > 0$ of different resources. We say that the tuple $P = (B, \bar{x}, u, n, c)$ is a *process* if $B \in \mathcal{B}$ (the *basic process* defining the behavior of the process), $\bar{x} = (x_1, \ldots x_m) \in \mathbb{R}_+^m$ (the *amounts* of resources owned by P), $u : \mathcal{P}(\mathtt{ACT}) \times \mathbb{R}_+^m \longrightarrow \mathbb{R}$ (the *utility function*), $n : \mathtt{ACT} \times \mathbb{R}_+^m \longrightarrow \mathbb{R}_+ \cup \{\infty\}$ (the *necessity* function), and finally $c : \mathtt{ACT} \times \mathbb{R}_+^m \longrightarrow \mathbb{R}^m$ (*consumption* of resources function).

Given a process P_i, we will usually consider $P_i = (B_i, \overline{x_i}, u_i, n_i, c_i)$, that is, indices will denote the process to which $B, \bar{x}, \ldots$ are related. □

The utility and necessity functions must fulfill some simple *rational* properties. We suppose that utility does not decrease if resources increase, that is, for any $A \in \mathcal{P}(\mathtt{ACT})$, and any $1 \leq i \leq m$, we have $\frac{\triangle u(A, \bar{x})}{\triangle x_i} \geq 0$. Given the fact that utility functions will be always applied to the initial actions of a behavior B, we suppose $u(\emptyset, \bar{x}) = 0$. This property means that a *deadlocked* process does not need any resources, so they will be shared by the rest of the processes (because those resources do not add any utility to the owner). In microeconomic theory, there are other restrictions that are usually imposed (strict monotonicity, convexity, etc) but they are not needed in our current framework (see [11,12] for more details). Regarding the necessity function, we will impose the condition that the function n is non-increasing with the number of resources, that is, for any $1 \leq i \leq m$ and $\mu \in \mathtt{ACT}$, we have $\frac{\triangle n(\mu, \bar{x})}{\triangle x_i} \leq 0$. Moreover, we will suppose that $n(a, \bar{x}) = n(\tau_a, \bar{x})$, that is, a visible action a needs the same resources that its internal counterpart

τ_a. Finally, let us comment on a *trick* that we will intensively use in the examples of Section 3. There are many situations where resources cannot be split. For example, it makes no sense to consider that a process *owns* half of a printer. We will indicate that half of a printer is so *useful* as no printer at all by using utility functions like $u(A,(x_1,\ldots,x_i,\ldots,x_n)) = f(A,x_1,\ldots,\texttt{trunc}(x_i),\ldots,x_n))$.

We will extend the operational transitions for behaviors (as defined in Figure 1) to processes. The idea is that a process may perform a transition $\xrightarrow{\mu}$ if its corresponding behavior so does, and the process has enough resources. Formally, if $P = (B,\bar{x},u,n,c)$ then we have the following rule:

$$(\text{PRO})\frac{B \xrightarrow{\mu} B' \ \wedge \ n(\mu,\bar{x}) < \infty \ \wedge \ c(\mu,\bar{x}) \geq \bar{0}}{P \xrightarrow{\mu} P'}$$

where $P' = (B', c(\mu,\bar{x}), u, n, c)$. Let us note that we have overloaded the symbol $\longrightarrow$: It is used both to denote transitions of behaviors and transitions of processes.

A *system* will be the parallel composition of a (finite, non-empty) collection of processes. Processes will communicate in two ways: Either by synchronizing in the execution of actions or by exchanging resources. Given the nature of the systems that we would like to describe, we have decided to use a relatively complex parallel operator. Specifically, our parallel operator is a simplification of [7] where m among n cooperation is not considered (similar proposals appear in [3,6]). The idea is that synchronizations are not restricted to be binary. The parallel operator will have a tuple of subsets of $\texttt{Act}$ as parameter: $(A_1,\ldots,A_n)$. So, if we have the parallel composition of $P_1,\ldots,P_n$, a process P_i will perform the actions in $\texttt{ACT} - A_i$ asynchronously; if $a \in A_i$ then any P_j such that $a \in A_j$ must synchronize with P_i in order to perform a. Similarly, exchanges of resources may involve more than two processes.

Definition 6. Let $A_1,\ldots A_n \subseteq \texttt{Act}$. A *system* S consists in the parallel composition of n processes $P_1,\ldots P_n$ synchronizing, respectively, in the set A_i. We denote the system S by $\|_n^{A_i} P_i$. We denote the set of systems by $\mathcal{S}$. □

From now on we will assume that systems are initially *compatible* with the available resources. That is, if we want to specify a system having the use of a vector of resources $\bar{r}$ by $S = \|_n^{A_i} P_i$ then the sum of the total quantity of resources initially assigned to $P_1,\ldots,P_n$ must be equal to $\bar{r}$, that is, $\sum_i \overline{x_i} = \bar{r}$. In the following we introduce the operational rules for systems. The first rule uses two predicates that will be defined in forthcoming definitions.

Definition 7. (*The Exchange Rule*). Let $S = \|_n^{A_i} P_i$ be a system, where for any $1 \leq i \leq n$ we have $P_i = (B_i, \overline{x_i}, u_i, n_i, c_i)$. The operational transitions denoting exchange of resources that S may perform are given by the rule:

$$(\text{Par1})\frac{\texttt{valid}(S,\mathcal{E}) \ \wedge \ \texttt{allowed}(S,\mathcal{E})}{S \overset{\mathcal{E}}{\rightsquigarrow} \|_n^{A_i} P_i' \left[\forall\ 1 \leq i \leq n :\ P_i' = (B_i, \overline{x_i} - \sum_j \mathcal{E}_{ij} + \sum_j \mathcal{E}_{ji}, u_i, n_i, c_i)\right]}$$

where $\mathcal{E} \in (\mathbb{R}_+^m)^{n*n}$. We denote by $\rightsquigarrow^*$ the reflexive and transitive closure of $\rightsquigarrow$. We say that S is a *local equilibrium*, denoted by $S \not\rightsquigarrow$, if there do not exist S' and $\mathcal{E}$ such that $S \overset{\mathcal{E}}{\rightsquigarrow} S'$. □

Intuitively, a transition $\|_n^{A_i} P_i \overset{\mathcal{E}}{\rightsquigarrow} \|_n^{A_i} P_i'$ indicates that P_i gives to P_j the quantities of resources indicated by $\mathcal{E}_{ij}$ and receives from it the quantities given by $\mathcal{E}_{ji}$. The total amount of resources that P_i owns is equal to its original resources minus the resources that it gives plus the resources that it receives. As we will see later, processes will not be allowed to perform actions until the system reaches a local equilibrium. The idea is that an equilibrium represents a *good* distribution of resources (because no more *useful* exchanges can be made) and so processes must be *delayed* until a local equilibrium is reached. If we allow a process P_i to perform transitions before the system is an equilibrium (for example, as soon as $n_i(a, \overline{x_i}) < \infty$ and $c(a, \bar{x}) \geq \bar{0}$ hold) then the system will not be working at its *best* possible performance(s).

Now we will define the predicates $\texttt{valid}(S, \mathcal{E})$ and $\texttt{allowed}(S, \mathcal{E})$. The first predicate holds if the processes do not give more resources that the ones that they initially own, and the diagonal of the matrix is filled with $\bar{0}$.

Definition 8. Let $S = \|_n^{A_i} P_i$ be a system with $P_i = (B_i, \overline{x_i}, u_i, n_i, c_i)$ for all $1 \leq i \leq n$. We say that $\mathcal{E} \in (\mathbb{R}_+^m)^{n*n}$ is a *valid exchange matrix for* S, denoted by $\texttt{valid}(S, \mathcal{E})$, if for all $1 \leq i \leq n$ we have $\sum_j \mathcal{E}_{ij} \leq \overline{x_i}$ and $\mathcal{E}_{ii} = \bar{0}$. □

Regarding the predicate $\texttt{allowed}(S, \mathcal{E})$ we have several possibilities. This choice would depend on the kind of policy[1] that we want to implement. As we said in the introduction we consider two possible policies. The first policy, the *preserving utility* policy, will allow exchange of resources only if, after the exchange, at least one process improves and no process gets worse. Intuitively, processes are the owners of the resources and they will not give up them if they do not receive a *compensation*. The second policy, the *maximizing utility* policy, will allow exchanges if the overall situation of the system improves. In order to measure the improvement of the system, we consider the sum of the utilities of all the processes.[2] As we commented in the introduction of this paper, such a policy could be interpreted by thinking that processes are not the *owners* of resources because they can be *expropriated* without any compensation. Nevertheless, this policy is more efficient than a totally centralized system of resources delivery because it avoids the return and redelivery of the same resources to a process. In economic terms, the first of these policies can be seen as a kind of *market economy* while the second one could be interpreted as a kind of *planned economy*.

Definition 9. Let $S = \|_n^{A_i} P_i$ be a system, where for any $1 \leq i \leq n$ we have $P_i = (B_i, \overline{x_i}, u_i, n_i, c_i)$, and let $\mathcal{E}$ be a matrix such that $\texttt{valid}(S, \mathcal{E})$.

The $\texttt{allowed}(S, \mathcal{E})$ predicate under a *Preserving Utility Policy* is defined as $\forall\ 1 \leq i \leq n$ we have $u_i(\texttt{Imm}(B_i), \overline{x_i}) \leq u_i(\texttt{Imm}(B_i), \overline{x_i} - \sum_j \mathcal{E}_{ij} + \sum_j \mathcal{E}_{ji})$ and $\exists\ 1 \leq k \leq n :\ u_k(\texttt{Imm}(B_k), \overline{x_k}) < u_k(\texttt{Imm}(B_k), \overline{x_k} - \sum_j \mathcal{E}_{kj} + \sum_j \mathcal{E}_{jk})$.

[1] The choice of a *good* policy is not a trivial task. Actually, it is impossible to choose a *perfect* policy. This problem is related with the social welfare aggregator problem. Arrow's *impossibility theorem* shows that there does not exist such an aggregator fulfilling a certain set of *desirable* properties (see [1] for more details).

[2] Using microeconomics terminology, this is a *Benthan* social welfare aggregator.

The $\texttt{allowed}(S, \mathcal{E})$ predicate under a *Maximizing Utility Policy* is defined as $\sum_i u_i(\texttt{Imm}(B_i), \overline{x_i}) < \sum_i u_i(\texttt{Imm}(B_i), \overline{x_i} - \sum_j \mathcal{E}_{ij} + \sum_j \mathcal{E}_{ji})$. □

From now on, if the results depend on the chosen policy, we will indicate under which one we are working. Once we have finished the definition of rule (Par1), let us remark that exchange of resources produces a lot of nondeterminism. For example, consider the processes P_{j_1} and P_{j_2} presented in the introduction. For any $a > 0$, and $2 \leq x \leq 4$, exchanges where P_{j_1} receives a units of the resource i_1 from P_{j_2} and P_{j_1} gives $x \cdot a$ units of the resource i_2 from P_{j_2} will be allowed.

Definition 10. (*The Synchronization Rules*). Let $S = \|_n^{A_i} P_i$ be a system. The operational transitions denoting the actions that S may perform are given by the rules:

$$\text{(Par2)}\ \frac{S \not\rightarrow \wedge\ P_j \xrightarrow{\mu} P_j' \wedge \mu \notin A_j}{S \xrightarrow{\mu} \|_n^{A_i} P_i'' \quad \left[\forall\ 1 \leq i \leq n:\ P_i'' = \begin{cases} P_j' & i = j \\ P_i & i \neq j \end{cases}\right]}$$

$$\text{(Par3)}\ \frac{S \not\rightarrow \wedge\ P_j \xrightarrow{a} P_j' \wedge a \in A_j \wedge \forall\ 1 \leq k \leq n:\ (a \in A_k) \Longrightarrow (P_k \xrightarrow{a} P_k')}{S \xrightarrow{a} \|_n^{A_i} P_i'' \quad \left[\forall\ 1 \leq i \leq n:\ P_i'' = \begin{cases} P_i' \text{ if } a \in A_i \\ P_i \text{ if } a \notin A_i \end{cases}\right]}$$

□

Let us note that we have overloaded the symbol denoting transitions. The condition $S \not\rightarrow$ indicates that the system has reached a local equilibrium. If S is not a local equilibrium, more useful exchanges can be made (i.e. the system may improve) and so actions should not be performed. Rule (Par2) is applied for interleaving actions. In this case, the only process that changes is the one involved in the transition. Rule (Par3) says that if a process P_j may perform an action a belonging to its synchronization set A_j, then all the processes having a in their synchronization sets must also perform a; all these processes will perform this action synchronously.

Finally, let us remark that our process algebra is a conservative extension of LOTOS: It is enough to consider that the functions u and n are always equal to 0 and that $c(\mu, \bar{x}) = \bar{x}$, for any $\mu \in \texttt{ACT}$.

2.1 Some Properties of the Language

In this section we study some properties that local equilibria fulfill according to the functions defining the involved processes. These properties will be used in the next section of this paper when analyzing the defined specifications. The first result says that any system may evolve into a local equilibrium.

Lemma 1. For any system S there exists S' such that $S \rightsquigarrow^* S' \not\rightarrow$. □

The proof of this result is immediate. Let us remark that if S is already a local equilibrium then $S' = S$ (that is, there is a $\rightsquigarrow^*$ derivation with length equal to zero). The following result states that once a local equilibrium has been reached, no process will keep resources that do not add utility to it if there exists another process that uses the resource.

Lemma 2. Let $S = \|_n^{A_i} P_i$. Let us suppose that there exist two processes P_i, P_j, $\epsilon > 0$, and $1 \leq r \leq m$ such that for any $\overline{x}$ we have

- $u_i(\mathtt{Imm}(B_i), \overline{x}) = u_i(\mathtt{Imm}(B_i), (x_1, \ldots, x_r - \epsilon, \ldots, x_n))$, and
- $u_j(\mathtt{Imm}(B_j), \overline{x}) < u_j(\mathtt{Imm}(B_j), (x_1, \ldots, x_r + \epsilon, \ldots, x_n))$.

Under the previous conditions, for any S' such that $S \rightsquigarrow^* S' \not\rightsquigarrow$ we have that $x'_{ir} \leq x_{ir} - \epsilon$, where x_{ir} denotes the amount of the resource r owned by P_i in S (similar for x'_{ir} and S').

Proof Sketch: By contradiction. Suppose $x'_{ir} > x_{ir} - \epsilon$. We have that, under both policies, there exists an exchange where P_i gives part of the resource r to a process P_k (there is at least a process increasing its utility by increasing the amount of r). So, S' is not a local equilibrium. □

An immediate corollary of the previous result is that if the condition holds for any ϵ such that $x_{ir} \geq \epsilon > 0$ then $x'_{ir} = 0$. Note that in the previous result is essential to suppose that another process increase its utility by increasing its amount of the resource r. The next result states that under a maximizing utility policy, all the local equilibria that can be reached from a system S have the same *global* utility.

Definition 11. Let $S = \|_n^{A_i} P_i$ be a system, where for any $1 \leq i \leq n$ we have $P_i = (B_i, \overline{x_i}, u_i, n_i, c_i)$. The *total* utility of S, denoted by $\mathtt{total}(S)$, is defined as $\sum_i u_i(\mathtt{Imm}(B_i), \overline{x_i})$. □

Theorem 1. Let $S = \|_n^{A_i} P_i$ be a system, where for any $1 \leq i \leq n$ we have that $P_i = (B_i, \overline{x_i}, u_i, n_i, c_i)$. Let S_1, S_2 be systems such that $S \rightsquigarrow^* S_1 \not\rightsquigarrow$, and $S \rightsquigarrow^* S_2 \not\rightsquigarrow$. If the *maximizing utility* policy is used then $\mathtt{total}(S_1) = \mathtt{total}(S_2)$.

Proof. Let $S_1 = \|_n^{A_i} P_{1i}$ and $S_2 = \|_n^{A_i} P_{2i}$, where for any $1 \leq i \leq n$ we consider $P_{1i} = (B_i, \overline{x_{1i}}, u_i, n_i, c_i)$ and $P_{2i} = (B_i, \overline{x_{2i}}, u_i, n_i, c_i)$. Let us suppose that $\mathtt{total}(S_1) > \mathtt{total}(S_2)$. Let $\mathcal{E}$ be an exchange matrix such that for any $1 \leq i \leq n$ we have $\sum_j \mathcal{E}_{ji} - \mathcal{E}_{ij} = x_{1i} - x_{2i}$ and $\mathtt{valid}(S_2, \mathcal{E})$. Given the fact that $\sum_i x_{1i} = \sum_i x_{2i}$, such a matrix fulfilling $\mathtt{allowed}(S_2, \mathcal{E})$ always exists. Therefore, there exists S' such that $S_2 \overset{\mathcal{E}}{\rightsquigarrow} S'$, which represents a contradiction. □

Note that different local equilibria may have different assignment of resources among processes; the previous result only assures that the sum of the utilities is the same, not that the composition of the baskets are equal. Note that this result does not hold under a preserving utility policy: Different equilibria do not have necessarily the same total utility.

3 EXAMPLES

In this section we will show how PAMR can be used to specify and analyze concurrent systems where resources play an important role. We will present two

classical examples: The dining philosophers and consumers/producers. Besides, we will study the absence of deadlock in these systems.

In the following we assume that an undefined value of a function is set to an arbitrary value. Actually, these cases will not be possible because they will correspond with a set of actions (resp. with an action) not reachable by the corresponding processes.

3.1 The Dining Philosophers

In this classical problem, five (male) philosophers stay by a dining table, which contains five forks. We will consider that these are the *resources* of the system: $fork_1, \ldots, fork_5$. These philosophers have only two tasks: To think and to eat. When a philosopher wants to eat, he must take both forks staying besides his dish (we suppose that philosopher i must take forks i and $(i \bmod 5) + 1$). The behavior of the philosophers can be described as:

$$Philosopher_i := think_i \,;\, eat_i \,;\, Philosopher_i$$

We suppose that the initial distribution of forks gives to philosopher i the *i-th* fork, that is, $\overline{x_i} = (\delta_{i1}, \ldots, \delta_{i5})$ where $\delta_{ij} = 1$ if $i = j$, and $\delta_{ij} = 0$ otherwise. Utility functions are defined as:

$$u_i(\{think_i\}, \bar{x}) = 0 \qquad u_i(\{eat_i\}, \bar{x}) = \mathtt{trunc}(fork_i) \cdot \mathtt{trunc}(fork_{(i \bmod 5)+1})$$

That is, if a philosopher wants to eat, he gets utility only if he owns both forks; otherwise, it will be the same to have one or none. Besides, a philosopher gets no utility by holding *useless* forks. We set $u_i(\{think_i\}, \bar{x})$ to an arbitrary value because no resources are needed to think. The necessity functions are defined as follows:

$$n_i(think_i, \bar{x}) = K_i \qquad n_i(eat_i, \bar{x}) = \frac{E_i}{u_i(\{eat_i\}, \bar{x})}$$

In this case, the time that philosophers spend thinking may be different, but does not depend on the resources. Besides, we also assume that they eat at different speeds. Finally, no resources are consumed by performing actions, so the consumption function is defined as $c_i(a, \bar{x}) = \bar{x}$. In conclusion, the system can be defined as

$$Dining\_Philosophers = \|_5^{A_i} P_i$$

where for any $1 \leq i \leq 5$, $A_i = \emptyset$ and $P_i = (Philosopher_i, \overline{x_i}, u_i, n_i, c_i)$.

The following result shows that this system cannot get deadlocked. As in the next example, the definition of the utility functions plays an important role in the absence of deadlocks.

Lemma 3. (*Absence of Deadlocks for Dining_Philosophers*). Consider a system S such that

$$Dining\_Philosophers \rightsquigarrow^* S_1 \xrightarrow{a_1} S_1' \rightsquigarrow^* S_2 \cdots \xrightarrow{a_n} S_n' \rightsquigarrow^* S$$

If S is not a local equilibrium then there exist $S', \mathcal{E}$ such that $S \overset{\mathcal{E}}{\rightsquigarrow} S'$; otherwise, there exist S', a such that $S \xrightarrow{a} S'$.

Proof. The first case is trivial from Lemma 1. If a philosopher is willing to think, this action may be performed (no resources are needed). Suppose that all the philosophers desire to eat, and none of them can. This implies that all of them have zero utility. Regardless of the chosen policy, S is not a local equilibrium, because any exchange where some philosopher takes his two forks is allowed (note that fractions of forks will not be exchanged because they do not increase utility). □

3.2 The Bounded-Buffer Producers/Consumers Problem

Consider n producers and m consumers. The former have access to a buffer where they can place their products; these products will be taken by the latter. The buffer is bounded: A consumer cannot get out a product if the buffer is empty and a producer cannot put in a product if the buffer is full. In order to avoid any damage in the structure, the buffer must be accessed preserving mutual exclusion. The behaviors of the involved processes are given by:

$$Producer := produce\ ;\ enqueue_1\ ;\ enqueue_2\ ;\ Producer$$

$$Consumer := dequeue_1\ ;\ dequeue_2\ ;\ consume\ ;\ Consumer$$

In order to visualize how mutual exclusion is implemented, the *enqueue* and *dequeue* operations are split into two different steps. The initial resources of the system are: a unit of *mutual exclusion*, zero *produced* units, and p free *places*. We assume that these *resources* are randomly distributed among the processes in such a way that $\sum_{i=1}^{n+m} \overline{x_i} = (1, 0, p)$. We will define a unique utility function for all consumers and producers:

$$u(\{enqueue_1\}, \bar{x}) = \begin{cases} 1 \text{ if } \mathtt{trunc}(x_1) \cdot \mathtt{trunc}(x_3) \geq 1 \\ 0 \text{ otherwise} \end{cases}$$

$$u(\{dequeue_1\}, \bar{x}) = \begin{cases} 1 \text{ if } \mathtt{trunc}(x_1) \cdot \mathtt{trunc}(x_2) \geq 1 \\ 0 \text{ otherwise} \end{cases}$$

$$u(\{enqueue_2\}, \bar{x}) = \begin{cases} 2 \text{ if } x_1 = 1 \\ 0 \text{ otherwise} \end{cases} \qquad u(\{dequeue_2\}, \bar{x}) = \begin{cases} 2 \text{ if } x_1 = 1 \\ 0 \text{ otherwise} \end{cases}$$

$$u(\{produce\}, \bar{x}) = K_1 \qquad u(\{consume\}, \bar{x}) = K_2$$

Let us note that the order structure of the buffer is not fully represented: The distinct products or distinct free places are not distinguished. Regarding the necessity function, it can be defined from the utility function as:

$$n(a, \bar{x}) = \frac{K_3}{u(\{a\}, \bar{x})}$$

Let us remark that if a process does not own the necessary resources, its utility function will return 0, and so, the necessity function will be equal to infinite (let us remember that we consider $\frac{r}{0} = \infty$). In this example, the last two resources are created and consumed, so we have to define the value of the consumption function in the appropriate way:

$$c(enqueue_1, \bar{x}) = (x_1, x_2, x_3 - 1) \qquad c(enqueue_2, \bar{x}) = (x_1, x_2 + 1, x_3)$$
$$c(dequeue_1, \bar{x}) = (x_1, x_2 - 1, x_3) \qquad c(dequeue_2, \bar{x}) = (x_1, x_2, x_3 + 1)$$
$$c(produce, \bar{x}) = \bar{x} \qquad c(consume, \bar{x}) = \bar{x}$$

Finally, the system may be specified as:

$$Bounded\_Buffer = \|_{n+m}^{A_i} P_i$$

where, for any $1 \leq i \leq n$ we have $P_i = (Producer, \overline{x_i}, u, n, c)$, and for any $n+1 \leq j \leq n+m$ we have $P_j = (Consumer, \overline{x_j}, u, n, c)$. Besides, for any $1 \leq k \leq n+m$ we have $A_k = \emptyset$.

One important property that we must have is *mutual exclusion*. The next result states that accesses to the buffer are done by preserving this property: Once a process performs an action belonging to $\{enqueue_1, dequeue_1\}$, it is assured that this process will perform the corresponding *second part* of the performed action ($enqueue_2$ and $dequeue_2$ respectively) before any other process performs an action belonging to $\{enqueue_1, dequeue_1\}$.

Lemma 4. Let S be a system such that

$$Bounded\_Buffer \rightsquigarrow^* S_1 \xrightarrow{a_1} S_1' \rightsquigarrow^* S_2 \xrightarrow{a_2} \cdots \rightsquigarrow^* S \not\rightsquigarrow$$

Let us suppose $S \xrightarrow{a} S' \rightsquigarrow^* S'' \not\rightsquigarrow$, where $a \in \{enqueue_1, dequeue_1\}$ and a has been performed by P_j (that is, $P_j \xrightarrow{a} P_j'$). For any transition $S'' \xrightarrow{b} S'''$ such that $b \in \{enqueue_1, dequeue_1, enqueue_2, dequeue_2\}$ we have that if $a = enqueue_1$ then $b = enqueue_2$ (resp. if $a = dequeue_1$ then $b = dequeue_2$) and this transition has been performed by the evolution of P_j' from S' to S''.

Proof Sketch: Let us consider the producers case (for consumers is similar). Under the *maximizing utility policy*, when a producer performs its $enqueue_1$ action, for any reachable local equilibrium, the unit of mutual exclusion resource is given again to that producer because no other process would have more utility than this one by owning it (note that this producer will have utility 2). Under the *preserving utility policy*, when a producer performs an $enqueue_1$ action, the mutual exclusion resource cannot be taken by another process because the utility of the owner would decrease. □

Lemma 5. (*Absence of Deadlocks for Bounded_Buffer*). Let us consider a system S such that $Bounded\_Buffer \rightsquigarrow^* S_1 \xrightarrow{a_1} S_1' \rightsquigarrow^* S_2 \cdots \xrightarrow{a_n} S_n' \rightsquigarrow^* S$. If S is not a local equilibrium then there exist $S', \mathcal{E}$ such that $S \overset{\mathcal{E}}{\rightsquigarrow} S'$; otherwise, there exist S', a such that $S \xrightarrow{a} S'$.

Proof Sketch: If there are no products then any producer will be able to produce. If there are no free places, any consumer will be able to consume. In both policies, the mutual exclusion resource will be taken (and owned) by a process only if it makes that process to perform an action. Therefore, at least one process will get the resources it needs. □

4 CONCLUSIONS AND FUTURE WORK

In this paper we have presented a formalism to specify systems where resources can be exchanged among subsystems. These exchanges will improve the performance of the system. We have studied some (theoretical) properties of the

language. Finally, we have specified and studied two examples where the characteristics of our language have been shown.

There are some interesting directions of research. For example, we could define a real-time extension of our language where the necessity functions play a fundamental role: They will induce delays in the behavior of processes. These delays can be either deterministic or can be defined by means of a random variable. In the latter case, we will consider some of the current models of stochastic process algebras (e.g. [2,4,8]). Besides, we will consider that the exchange of resources takes time. So, sequences of $\rightsquigarrow$ transitions should be grouped. Another interesting point is that utility functions (as well as necessity and consumption functions) are *static*, that is, they do not change along the performance of the system. Once time is taken into account, it is straightforward to extend the previous framework to consider *dynamic* functions. Finally, the relation between our framework and management of software projects should be explored. Indeed, the tasks of a project can be seen as processes while the members of the project can be seen as the resources *used* by the tasks.

References

1. K.J. Arrow. *Social Choice and Individual Values*. Wiley, 2nd edition, 1963.
2. M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, 1998.
3. T. Bolognesi. A graphical composition theorem for networks of LOTOS processes. In *10th International Conference on Distributed Computing Systems*, pages 88–95. IEEE-CS Press, 1990.
4. M. Bravetti and R. Gorrieri. The theory of interactive generalized semi-markov processes. To appear in Theoretical Computer Science, 2001.
5. P. Brémond-Grégoire and I. Lee. A process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, 189(1-2):179–219, 1997.
6. J. Davies and S. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.
7. H. Garavel and M. Sighireanu. A graphical parallel composition operator for process algebras. In *Formal Description Techniques for Distributed Systems and Communication Protocols (XII), and Protocol Specification, Testing, and Verification (XIX)*, pages 185–202. Kluwer Academic Publishers, 1999.
8. H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. To appear in Theoretical Computer Science, 2001.
9. ISO/IEC. ODP Trading Function. Draft International Standard 13235, ISO - Information Processing Systems, 1995.
10. N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. *6th ACM Symp. on Principles of Distributed Computing*, 137–151, 1987.
11. A. Mas-Colell, M.D. Whinston, and J.R. Green. *Microeconomic Theory*. Oxford University Press, 1995.
12. M. Núñez. Including microeconomic theory into FDTs: A first approach. Available at: `http://dalila.sip.ucm.es/~manolo/papers/exchange.ps.gz`, 2000.
13. M. Núñez and I. Rodríguez. `PAMR`: A process algebra for the management of resources in concurrent system. Available at: `http://dalila.sip.ucm.es/~manolo/papers/pamr.ps.gz`, 2001.

The Design of a Cellular Network for Real-Time Auction*

Pedro S. Rodríguez-Hernández, Francisco J. González-Castaño, José M. Pousada-Carballo, Manuel J. Fernández-Iglesias, and Jaime García-Reinoso

Departamento de Ingeniería Telemática, Universidad de Vigo.
{pedro,javier,chema,manolo,reinoso}@det.uvigo.es

Abstract. Auctions are an ancient form of economic activity and, as such, have evolved with technology. However, some forms of traditional auction have survived unchanged, even in areas of great economic interest, until modern regulations imposed a change. We present the design of a specialized cellular network that has recently replaced traditional fish auctions in Galicia (Spain).

1 Introduction

Galicia (Spain), like other Atlantic regions, obtains a large percentage of its income from sea industries: Inshore and deep-sea fishing, canning industries and shipyards. One of the main Galician cities, Vigo, is the most important fishing port in Europe. Inshore Galician fishermen are organized in guilds (*cofradías*), which sell captures in public auctions immediately after inshore boats are unloaded. These guilds have annual fish sales in the range of 100-200 million Euros [27].

A traditional fish auction takes place in a large hall where captures are displayed on the floor in boxes, classified by species, quality and weight. Authorized dealers and public walk around, listening to auction leaders who sing box prices in decreasing order. The first dealer who makes an indication to stop the countdown obtains the box.

European Union food handling regulations have recently interfered with tradition. It is compulsory to establish a physical barrier between displayed fish and dealers. In some places, such as Plymouth Fish Trade (UK) or Ostend Fish Auction (Belgium), dealers sit in desks with bid buttons, in dedicated auction rooms. Prices are displayed on a countdown electronic clock [21].

In the case of Galicia, this solution is not desirable, for two reasons:

Tradition: For centuries, dealers have been allowed to move around. Mobility is, therefore, a major concern.
Tourism: Inshore fish auctions are important tourist destinations at Galician sea ports. Any solution should have little impact on this secondary profit source, by introducing as few changes as possible in auction rooms.

* Supported by Consellería de Industria, Xunta de Galicia (Spain) and SAEC-DATA S.A. (Spain)

As a result, the Galician government decided to fund a system in which dealers carry a mobile hand held computer to make their bids, and no major changes are introduced in the auction room. In addition:

A. The system must respect dealer privacy: dealers do not necessarily need to identify themselves at the control booth. Also, terminals must be operative when hidden.
B. Terminals must offer balance status and other messages when required by the user.
C. The system must be robust: all bids must arrive to their destination. At the same time, only authorized dealers and (more important) dealers with positive balance are allowed to bid.
D. Involuntary or hostile interference may disable the system, but must not cause a transition to an incoherent state.
E. Temporary terminal disconnection, either voluntary (leaving the auction room) or not (fading), must be transparent to users, unless they want to bid at the same time.
F. All dealers should be free to move to a different auction place (up to 64 possible guild facilities), without need of new registration.

This paper describes a cellular system that satisfies requirements A-F, which has been jointly developed by Universidad de Vigo and SAEC-DATA S.A. (Spain).

2 Background: Electronic Auctions

By electronic auction me mean any computer-related auction technology. For a review of most of these technologies, see [21]. We can make the following classification:

1. *Automated auctions*. Users are present at an auction room. Partially or totally, the auction process is replaced by electronic aids. This is the case of the European fish auctions described in the previous section. Other examples are the Dutch flower industry [26] and the Nigerian coffee exchange [21].
2. *Internet auctions*: Users are not located in an auction room. The auction is implemented as a client-server distributed system over the Internet. Users make their bids through a WWW browser [7, 25].
3. *Artificial intelligence auctions*: For research purposes, real entities (such as users) are replaced by computer processes [20].

Current systems for automated auction do not fulfil requirements A-F. As far as the authors know, no bidirectional mobile terminals for real-time auction were available in the market before this project. Existing terminals just transmit a code when a button is pressed, which is used to stop the auction. In some cases, such as infrared terminals, the underlying technology is intrinsically unidirectional[1]. Obviously, bidirectionality is required to transmit messages to the

[1] One of the Galician auctions, Portonovo, used this kind of terminal. Portonovo has recently adopted our system.

user. However, there are more subtle issues that depend on bidirectionality. For example, it is possible to implement advanced protocols for channel sharing that guarantee that all terminal transmissions will eventually reach their destination.

A bidirectional system also permits to establish different working modes. In our case, at least, there are two: terminal recognition, to let the system know which terminals are present (equivalent to GSM self-identification signalling phase), and auction (equivalent to GSM traffic exchange phase) [17].

3 A Mobile Network for Automated Auction

Our model is a particular case of automated auction. In order to satisfy constraint F above, our system is inspired by a GSM network. Each user owns a mobile terminal with a unique serial number. When the user enters a new auction room, the terminal initiates an automatic dialog with the base station placed there, and obtains a local working code. All cells are linked via the Internet, for administrative and roaming purposes.

Figure 1 shows the electronic entities that participate in the auction.

- *Cell*: A cell is an auction room. All active terminals entering a cell are assigned to it.
- *Base station*: The core of the bid system. It is responsible of assigning TDM slots to user terminals, resolving conflicts, recognizing present terminals, transmitting messages to them, and deciding who wins an auction.
- *Terminal*: A terminal is a mobile hand held computer that sends bids to the base station and receives messages from it.
- *Clock computer*: During auctions, it displays a price countdown. A terminal bid stops the clock computer when a user is interested in the current price. Also, the clock computer:
  - Sets terminal recognition mode or auction mode in the base station.
  - Transmits a list or valid users and their balance to the base station.
  - Receives a list of present users from the base station.
  - Updates user balances.
  - Generates messages that are passed to the base station, to be transmitted to terminals.

4 A discussion on existing technologies

In this section, we discuss the advantages and drawbacks of three candidate wireless technologies to implement the modems of the base station and the terminals: Bluetooth, IEEE 802.11 Wireless LAN (WLAN) and 433Mhz band modems.

4.1 Bluetooth

Bluetooth (BT) [2] is a short-range wireless technology. The typical separation between devices is 10 m [28]. It has been designed for applications such as wireless headphones and computer-to-peripheral communications.

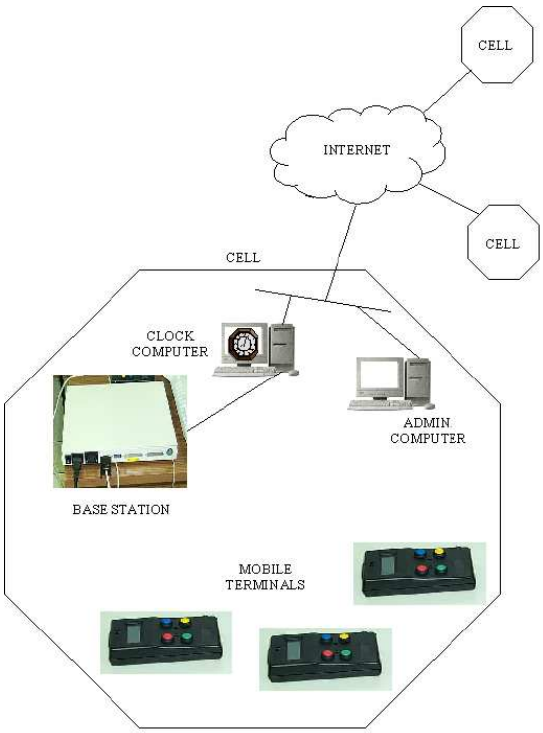


Fig. 1. System architecture

A BT network is composed of *piconets*. A piconet is a spontaneous network whose configuration changes when a BT device enters or leaves its range. A piconet has one master and up to seven *active* connected slaves [2].

It is also possible to create *scatternets*. A scatternet is a net of piconets. In a scatternet, the master of a piconet could be a slave in another piconet. In this scheme, inter-piconet communication is allowed [10].

Drawbacks: due to project specifications, there will be up to 250 users in the auction place. This place is small enough to fit into a BT piconet. Obviously, the only way to achieve that number of users is by defining a scatternet. This means that many user terminals will be both masters and slaves. According to our experience [10], the resulting configuration would be too complex, and would require further study to determine feasibility.

Advantages: from our point of view, the main advantages of this technology are its low power consumption ($\sim$ 1 mA standby and $\sim$ 60 mA peak) [15], small modem size (1 cm^2) and future low cost (5-10 USD) [11, 14, 24, 22, 4]. However, at the time this paper was written, BT modems were still far from that price goal.

4.2 WLAN

IEEE 802.11 Wireless LAN architecture [12, 13, 3, 23] is composed of a number of base stations (access points) and up to 2,048 terminals per base station. Transfer rates vary from 1 Mbps to 11 Mbps.

This speed range allows extremely low response times for auction applications. Nevertheless, current implementations impose some limits:

- In order to guarantee DSSS 11 Mbps performance, there must be a separation of 25 MHz between channels [8, 5]. The number of 5-MHz channels in the 2.4 GHz DSSS band varies from 13 (channels 1-13, ETSI) to a single one (channel 14, MKK) [13]. As intermediate cases, there are two contiguous channels in the Spanish regulatory domain and four contiguous channels in the French one (Spain and France have a joint population of 100,000,000 citizens, a considerable fraction of the EU market). As a result, there can be up to three overlapping cells in the same area with full performance.
- The typical number of terminals per access point in mass-market implementations is 64 (see table 1).
- In some cases, the manufacturer recommends less than 15-20 simultaneous transmissions per access point, to ensure the quality and performance of data transmission [18].

Company	AP	# users/AP	AP Price	WLAN cards	Card price
3COM	3CRWE74796B	63	825 USD	WLAN PC Card	179 USD
NOKIA	A032 AP	64	999 USD	C110	320 USD
Apple	AirPort Base Station	10	299 USD	AirPort Card	99 USD
Netgear	ME102	30 - 70	231 USD	MA401	84.95 USD
Xircom	APWE1120	64	298.99 USD	CWE1120	138.99 USD

Table 1. Survey of WLAN products

Drawbacks: using DSSS technology, a number of 250 modems per auction room requires cell planning. WLAN modems are too expensive for a simple bid terminal: according to table 1, we would need four Netgear access points to handle 250 MA401 cards (note that this would not satisfy the channel separation recommendation in [8, 5]), with a total cost of 22,162 USD. Also, at a transfer rate of 1 Mbps, typical power consumption is $\sim$ 10 mA standby and $\sim$ 400 mA peak [15], which is much higher than the requirements of BT or 433 MHz band modems (see section 4.3).

Advantages: the theoretical number of users in the WLAN standard fulfils our project specifications. The transfer rates are very interesting for high-performance implementations.

4.3 433 MHz band modems

The 433 MHz radiocontrol band is available in several EU countries for short-range[2] control purposes [6], and is equivalent to the 418 MHz UK band. Existing 433 MHz modems provide a transparent path for either narrowband modulations or square waves. As an example, the Radiometrix bidirectional RF unit, or BiM [19] has a peak transmission rate of 40 Kbps.

Drawbacks: the transfer rate is 275 times slower than WLAN rate. It is necessary to develop application-specific MAC drivers (which are part of WLAN card distributions).

Advantages: bidirectional 433 MHz modems are cheap ($\sim$ 35 USD) and small (4 cm^2). In the 250-terminal example above, total cost would be 8,785 USD (including a single modem for the base station). Power consumption is extremely low $\sim$ 5 mA standby and $\sim$ 10 mA peak [19].

4.4 Technology choice

We decided to discard Bluetooth, due to the complexity imposed by the seven-slave boundary [10].

Current WLAN hardware costs are too high for simple embedded systems. System cost is a major concern, because the auction environment is extremely aggressive, and it is necessary to replace mobile terminals and base stations quite often. Also, WLAN may require cell planning, as the number of users grow.

Finally, due to the previous reasons, we decided to implement an auction protocol using 433 MHz modems. In the following sections, we prove that it is possible to implement an efficient auction system on that platform, while keeping costs at the same level as previous unidirectional systems [21].

4.5 Control

We implemented system control on a Motorola MC68HC11 microcontroller [16], which executes the automated auction protocol. The MC68HC11 comprises an 8-bit CPU and timing and serial port peripherals. Byte-level auction protocol is supported by the Serial Communications Interface (SCI), via exceptions. An underlying NRZ source coding (with one start bit and one stop bit per byte) is injected in the RF module.

Figures 2 and 3 show an internal view of the base station and the mobile terminal, respectively, as implemented with a BiM modem and a MC68HC11 microcontroller. The mobile terminal has a LCD display for user messages (see figure 1), and four buttons for bidding, user message interfacing and set-up. The terminal has an approximate autonomy of one month with a DC 9V battery, in normal operating conditions.

[2] $\sim$ 30 m indoors, $\sim$ 100 m outdoors.

Fig. 2. Base station prototype

Fig. 3. Mobile terminal prototype

5 Automated Auction Protocol

Figure 4 shows the basic unit of the auction protocol Time Domain Multiplex (TDM). It has a $B \rightarrow T$ channel, which is transmitted from the base station to the terminals, and one smaller channel per auction in the opposite sense, $T \rightarrow B_1$ and $T \rightarrow B_2$[3]. We chose this structure due to the simplicity of our RF module. Other mobile systems with a complex RF interface, such as GSM, have a different TDM per transmission sense, in separate carriers [9].

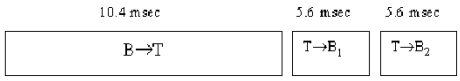


Fig. 4. Basic protocol unit

All auction protocol frames have the same structure: a 3-msec. synchronization burst (10101...), an idle byte, a frame start byte, and a data load (2 bytes for $T \rightarrow B$ and 17 bytes for $B \rightarrow T$). Here, a "byte" is a source-level byte (10 bits), including start and stop bits. A $T \rightarrow B$ frame is any frame placed in channel $T \rightarrow B$ (a $B \rightarrow T$ frame is defined accordingly). Overall frame length is 10.4 msec for $B \rightarrow T$ and 5.6 sec for $T \rightarrow B$[4], at 31250 Kbps. Note that a large percentage of transmission time is consumed by synchronization, which is imposed by the BiM RF module. All terminals in the same auction compete for the same $T \rightarrow B$ channel, in a slotted ALOHA fashion [1]. Collisions are detected as serial transmission error exceptions. In order to locate the correct $T \rightarrow B$ channel for an auction, terminals use $B \rightarrow T$ frame starting time as a reference (the base station injects a new $B \rightarrow T$ frame immediately after the last $T \rightarrow B$ channel).

Depending on frame type, there are two different data loads:

a) A $T \rightarrow B$ user frame has a two-byte data load. The first byte is the user local code. The second one is a checksum, for robustness.
b) A $B \rightarrow T$ base station frame has a 17-byte data load, which is shown in figure 5.

$B \rightarrow T$ $MODE$ bits control auction working mode:

- Let $a = 1$ or 2. $I_a = 0$, $L_a = 0$ sets terminal recognition mode for auction a. In this mode, The set V of valid terminals[5] is scanned by placing their

[3] Different fish auctions may take place in the same auction room. In the system described there are two auctions, but the generalization is straightforward.
[4] There exists an additional 1-msec guard before each channel.
[5] A cell can handle up to $size(V)$ terminals, $size(V) < N$, where N is the number of users in the network. V is actualized either by roaming or at the administration computer (see figure 1). In our case, $size(V) = 1000$ and $N = 65280$.

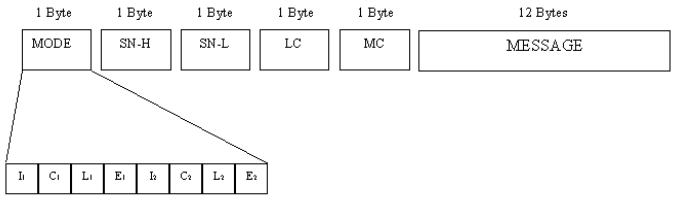


Fig. 5. $B \rightarrow T$ data load

serial number in the 16-bit SN field. A temporary local code is proposed for terminal SN in field LC. If terminal SN is present, it must acknowledge with a $T \rightarrow B_a$ frame containing LC. User terminals are disabled until they receive their (SN, LC) pair. As a result of terminal recognition, up to 250 terminals may participate in auctions that take place in the same cell. $E_a = 1$ activates auction a. Let R be the set of enabled terminals, $R \subset V$. If a user in R pushes bid button a, the terminal sends a bid by placing its LC in a $T \rightarrow B_a$ frame.

- $I_a = 1$, $L_a = 0$ and $SN - H = 0xFF$ sets winner identification mode in case of bid collision, for auction a. In this mode, the base station sends an upper bound UC_a for LC[6]. All potential winners such that $LC < UC_a$ must answer by placing their LC in a $T \rightarrow B_a$ frame. In case of a new collision, the process is repeated until a unique winner results. To achieve this goal, the base station varies UC_a in a logarithmic search [1] within R, starting from $\lceil \frac{size(R)}{2} \rceil$.
- $I_a = 1$, $L_a = 1$ and $SN - H = 0xFF$ sets winner acknowledgement mode for auction a. In this mode, the base station sends the winner LC in UC_a.
- $I_a = 0$, $L_a = 1$ sets purchase mode for auction a. By repeatedly pressing his bid button, the winner requests as many boxes as $T \rightarrow B_a$ frames. Each request must be acknowledged by a $B \rightarrow T$ purchase acknowledgement frame $(C_a = 1)$.

The auction protocol includes the possibility of sending ASCII messages to the terminals, to be shown on their LCD displays. Typical messages are current account balance and the total amount of boxes purchased. Any $B \rightarrow T$ frame can be used to send a 12-byte message (see fig. 5) to any terminal, by setting $MC = LC$. $MC = 0xFF$ sets broadcast mode.

6 Protocol Discussion

Bidirectional handshake ensures that all bids arrive to their destination (constraint C). It could be argued that a unidirectional system can achieve this

[6] In this mode, $SN - L$ and LC are UC_2 and UC_1, respectively.

objective, by means of repeated trials. However, our system has a *guaranteed* finite bid time for a given size of set R, which is a basic characteristic of real-time systems. Also, although a unidirectional system may filter users with negative balance or unregistered ones, a bidirectional system blocks RF transmission from them, which yields a better spectrum utilization.

Concerning constraint A, RF technology (as opposed to infrared) allows terminals to be operative when hidden. Dealers do not need to identify themselves at the control booth: the registration process is automatically handled by the terminal recognition procedure.

Our digital protocol design is more expensive than trivial solutions based on dumb terminals that simply send "pure" carriers. However, these systems can be easily fooled by interference, and an incoherent state may result violating constraint D. This is specially dangerous in the case of the 433 MHz band, which is widely used, for example, by construction cranes.

Constraint E has been considered for those cases in which users leave temporarily the auction room, or move to a place with high attenuation (behind a column, for example). As a design parameter, a terminal considers that its LC is no longer valid if it does not receive a $B \rightarrow T$ frame for 2 minutes. Consequently, the base station enters terminal recognition mode periodically. In order to satisfy constraint F, this mode is complemented with a roaming system, which keeps coherence of sets V across the cellular network. The system uses Internet for this purpose (see figure 1).

Finally, constraint B is supported by the messaging system.

7 Protocol Performance

Human perception is an essential consideration to evaluate protocol performance. Ideally:

- Elapsed time T_s between any winner bid and auction stop should be zero.
- Different bid times should be discriminated with arbitrary precision.
- Elapsed time T_w between auction stop and winner determination should be zero.

These goals are technologically impossible. However, it is possible to take advantage of the following subjective facts:

- An average human can hardly distinguish two manipulations separated by 0.1 sec. This fact is used to set price persistence time, 0.25 sec.
- Once auction stop is signaled, winner determination can be delayed an extra 1 sec. (this delay can be longer, if the stop alarm includes a "slow" message display on the auction screen).

With these facts in mind, protocol performance can be evaluated, as follows:

- Even in the case of collision, the system detects a bid within a basic protocol unit timeframe (figure 4). Therefore, $T_s \cong 20$ msec. For the same reason, non-simultaneous bid times can be discriminated with 20-msec precision.

– Elapsed time between auction stop and winner determination is variable, because it depends on the existence of collisions. In the best case there is only one winner, and $T_w = 0$ msec. In the worst case[7], and assuming $size(R) = 256$, $T_w \cong 20 \times log_2\lceil size(R)\rceil \cong 160$ msec.

Therefore, in this situation, $T_s + T_w \cong 180$ msec, which is much less than the initial objective of 1 sec for T_w alone.

In normal operating conditions some collisions will take place. In order to estimate real performance, the following assumptions were made: for a current price start time, some users try to stop the auction with response times that follow an exponential distribution with 100-msec mean. Potential winner LC identifiers are uniformly distributed in R. A collision takes place if there are at least two bids in the 20-msec window that includes the first bid (or, in other words, if two or more terminals try to use the same $T \rightarrow B$ channel). Then, a logarithmic search starts to find the winner, and T_w is equal to the number of 20-msec steps.

50,000 runs were made to calculate each point in figures 6 and 7. For different R sizes, a different number of simultaneous bids take place (shown as percentages of R size). Figures 6 and 7 show average and standard deviation of $T_s + T_w$, respectively (again, note that *100 msec was a goal for T_s alone*, which has a fixed length of 20 msec). In all cases, it was concluded that protocol performance was acceptable from a subjective point of view.

Alternatively, a cheaper unidirectional system was evaluated. In it, the terminals just send a $T \rightarrow B$ frame when the user presses the bid button, and the base station is a simple receiver. To be more precise, the terminals send a frame as soon as the bid button is pressed, and a second one with an uniformly distributed delay in [0,180] sec (the upper bound was chosen to let this system be as competitive as possible compared to ours). Again, some users try to stop the auction with response times that follow an exponential distribution with 100-msec mean. Figures 8 and 9 show average and standard deviation of T_s, respectively, across 50,000 runs. In this case, T_s is the elapsed time between the first user bid and the reception of the first correct $T \rightarrow B$ frame. Therefore, $T_w = T_s$. Note that the first user bid is not necessarily the winner one, since this system can not solve interference situations. An interference takes place if two user bid frames overlap.

Apparently, in figures 8 and 9, T_s seems to be lower than the ideal goal of 0.1 sec. However, as the percentage of simultaneous bids grow, all bids are potentially lost. To illustrate this problem, figure 10 shows the region in which 95% of the runs finish with a winner (*95%-customer satisfaction* region). Note that, for 50 users, which is a common number in fish auctions, the system becomes useless if more than 5 users make a simultaneous bid. The points in figures 8 and 9 that lie outside the 95%-customer satisfaction region are meaningless.

[7] A full logarithmic search is required. For example, in case of simultaneous bids from $LC = 0$ and $LC = 1$.

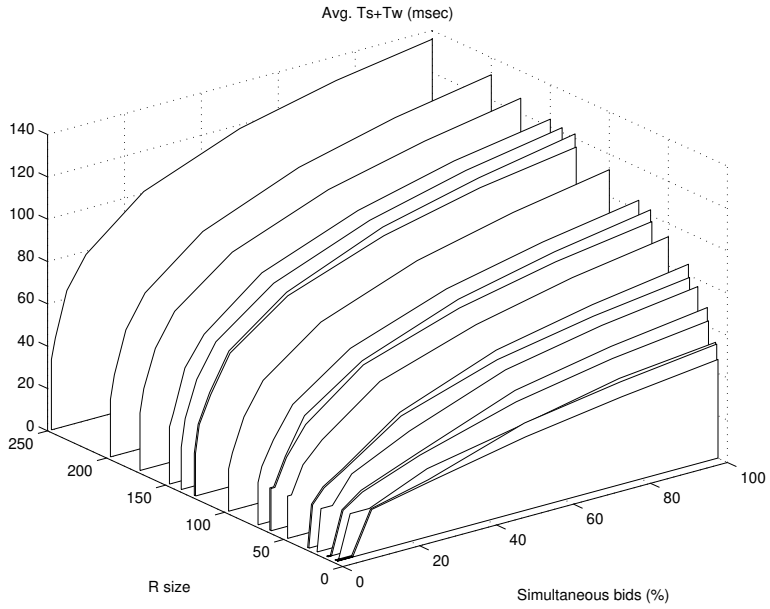


Fig. 6. Average $T_s + T_w$ (msec), bidirectional system

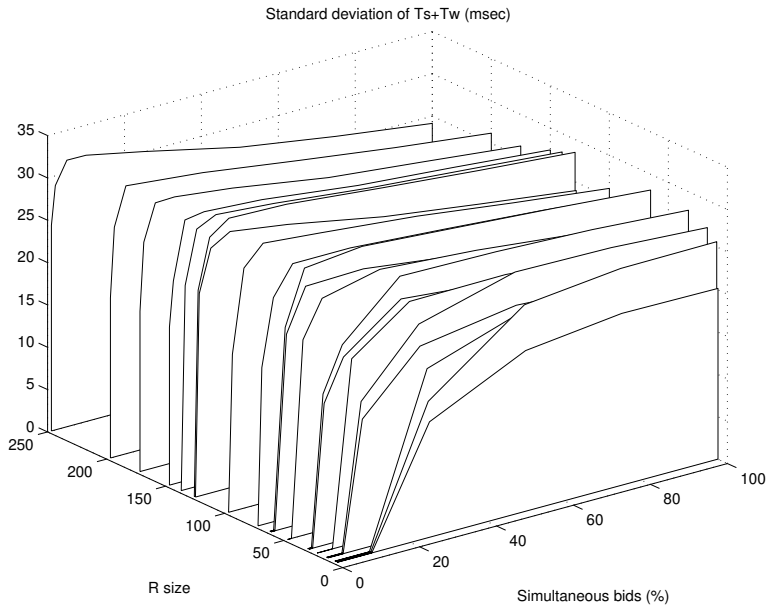


Fig. 7. Standard deviation of $T_s + T_w$ (msec), bidirectional system

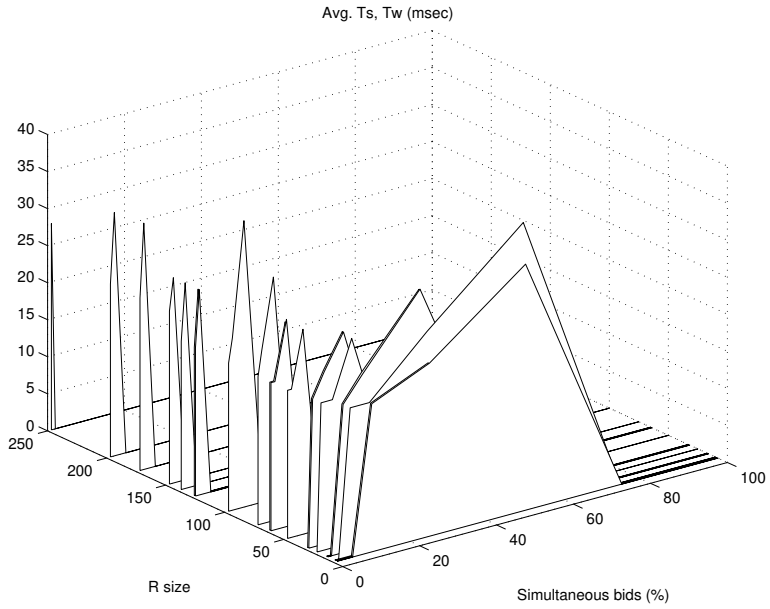


Fig. 8. Average T_s, T_w (msec), unidirectional system

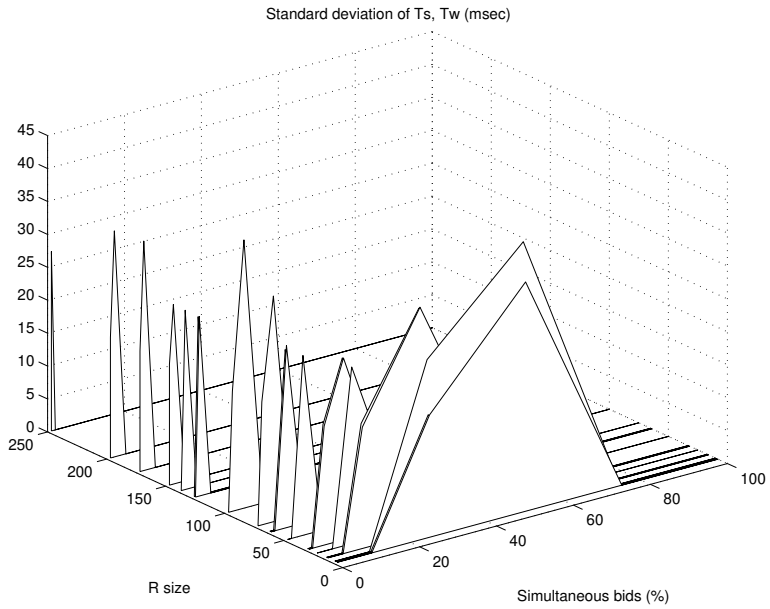


Fig. 9. Standard deviation of T_s, T_w (msec), unidirectional system

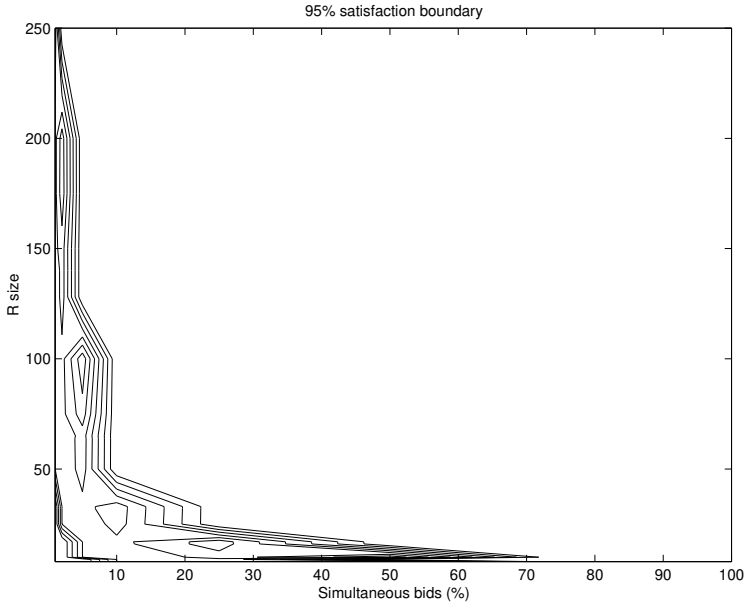


Fig. 10. 95%-satisfaction boundary, unidirectional system

8 Concluding Remarks

This paper describes a specialized cellular network for mobile auction, which is fully operational in the inshore guild system of Galicia, Spain. It has been proved that this system respects current regulations, economic considerations and tradition. At the time this paper was written, several cells were already connected to the network, with more than one thousand users.

Future research is oriented in two directions. First, protocol upgrade: future modifications, which are unknown at this moment, will be supported by the message field in $B \rightarrow T$ frames. On the other hand, we are currently working in extensions for on-line and off-line Internet and WAP auctions.

The authors wish to acknowledge the contribution of Enrique Pérez-Barros (SAEC-DATA S.A.) in system specification. All ideas in it are subject to pending patent P200002671 (Spain, to be extended).

References

1. Bertsekas, D. and R. Gallagher, Data Networks, Prentice-Hall International Ed., 1992.
2. Bluetooth SIG, Specification of the Bluetooth System-Core v1.0B, december 1999.
3. Bromax Wireless LAN FAQ, http://www.bromax.com.tw/faq.htm.
4. Casira, http://www.csr.com/options.htm, 2001.

5. Colubris Networks, Colubris Networks Wireless LAN Router, Configuration Scenario Guide, first edition, 2001.
6. Dirección General de Telecomunicaciones, España, Cuadro Nacional de Atribución de Frecuencias, 1991.
7. Ebay. Home page at http://www.ebay.com, 2001.
8. Ericsson Wireless Lan Technology for Mobility, Performance and Security. http://www.ericsson.com/wlan/te-80211.asp, 2001.
9. ETSI, "Digital cellular telecommunications system (phase 2+); Multiplexing and multiple access on the radio path (GSM 05.02 version 5.4.1)". ETSI technical report ETR 300 908, 1997.
10. García-Reinoso, J., J. Vales-Alonso, F. J. González-Castaño, L. Anido-Rifón and P. S. Rodríguez-Hernández, "A New m-Commerce Concept: m-Mall", Lecture Notes in Computer Science, in press.
11. GSMBox, http://uk.gsmbox.com/news/mobile_news/all/19304.gsmbox, 2001.
12. IEEE, IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Nov. 1997, P802.11.
13. IEEE, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band, Sep. 1999, P802.11b
14. Journaldunet, http://solutions.journaldunet.com/0103/010307inventel.shtml, 2001.
15. Lansford, J. and P. Bahl, "The Design and Implementation of HomeRF: A Radio Frequency Wireless Networking Standard for the Connected Home", Proc. IEEE 88, 2000, 1662–1676.
16. Miller, G. H, Microcomputer Engineering, Prentice Hall, 1993.
17. Mouly, M. and M. B. Pautet, The GSM system for mobile communications, ISBN 2-9507190-0-7, 1992.
18. Nokia, http://www.nokia.com/networks/wireless_lan/a032_faq.html, 2001.
19. Radiometrix, BiM page at http://www.radiometrix.co.uk/products/bimsheet.htm, 2001.
20. Rodríguez-Aguilar, J. A., F. J. Martín, P. Noriega, P. García, and C. Sierra, "Towards a Test-bed for Trading Agents in Electronic Auction Markets", AIComm 11(1), 1998, 5–19.
21. Schelfhout Computer Systems. Home page at http://www.schelfhout.com/ENG/E-View.htm, 2001.
22. Security-informer, http://www.security-informer.com/english/crd_bluecore01_209282.html, 2001.
23. Tay, Y. C. and K. C. Chua, "A capacity analysis for the IEEE 802.11 MAC protocol", Wireless Networks 7(2), 2001, 159–171.
24. Transfert, http://www.transfert.net/fr/revue_web/article.cfm?idx_rub=94&idx_art=285, 2001.
25. Ubid. Home page at http://www.ubid.com, 2001.
26. VanHeck, E. and P. M. Ribbers, "Experiences with Electronic Auctions in the Dutch Flower Industry", Electronic Markets vol 7(4), 1997.
27. Xunta de Galicia, Libro Blanco sobre la Gestión de las Cofradías de Pescadores de Galicia, Price Waterhouse-SACE UTE, 1994.
28. 10meters.com, http://www.10meters.com/blue_802.html, 2001.

Consistency Protocols in Globdata*

Francesc D. Muñoz-Escoí, Luis Irún-Briz, Pablo Galdámez, José M. Bernabéu-Aubán, Jordi Bataller, and M.Carmen Bañuls

Instituto Tecnológico de Informática.
Polytechnic University of Valencia.
Camino de Vera, s/n. 46022. Valencia. Spain.
E-mail: {fmunyoz,lirun,pgaldam,josep,bataller,banuls}@iti.upv.es

Abstract Globdata is a software tool that provides an object-oriented view of a set of replicated relational databases. These databases may be distributed in a wide area environment. To ensure consistency among the independent databases that build this environment some protocols are needed. These protocols have to reach a high degree of transaction completions, aborting a minimal percentage of them, and ensuring the consistency of all the databases.
A system of this kind may be used in the development of applications for companies that have several branch offices, such as banks, hypermarkets, etc. Usually, these companies have a large amount of operations that can be solved with local information, but sometimes the information generated in other branches is also needed. The services provided by Globdata allow an efficient completion of both kinds of requests.

1 Introduction and Motivation

When an organization starts an internet project, a variety of objectives must be met. Many of the current internet applications (i.e., e-bank applications) manage huge amounts of information. This information is mainly accessed with a strong geographical locality. In addition, these types of application usually strive for a high degree of availability, since they offer services not only to external, but also to internal clients, which must be capable of accessing the information at any time. The locality of the accesses suggest that in many cases the database can be partitioned [12,4]. In many scenarios, it may be necessary to replicate the information in a set of servers, each one attending its local clients. The different replicas of the database must be then interconnected, a WAN being usually the best fit alternative.

Another example of this scenario can be found in telephony applications, managing large amount of information, where access patterns are highly local.

When the databases containing the information must be replicated, it becomes necessary to introduce protocols and algorithms that provide a minimal set of guarantees about the consistency of the data [1,13]. The traditional approaches for replicating databases are centered in the use of fast LAN's

* This work has been partially supported by European grant IST-1999-20997

[9,10,8,6,5], where network-intensive protocols are used. In internet applications, the network is a limited resource, and the problems introduced by the WAN must be appropriately dealt with [11,3,2].

The Globdata project [7] strives to provide a solution for these kinds of applications in which efficiency, availability and high volume data handling must be achieved. It does so by defining a specific architecture for a set of replicated databases, together with a programming API and a set of consistency modes for data access.

This paper presents a proposal of protocols capable of meeting the consistency requirements posed by Globdata's goals. Although several protocols with slightly different goals are presented, all of them share a characteristic: They cannot be classified as pessimistic, since transactions are allowed to proceed locally, being checked for consistency violations at commit time. When a consistency violation is found, the transaction is rolled back.

The rest of the paper is organized as follows. Section 2 provides some common concepts shared by all consistency protocols, and several characteristics of the Globdata environment where the protocols are used. Section 3 presents an overall version of all consistency protocols and discusses the algorithm choices that can be adopted and the resulting algorithms, once these options have been chosen. In section 4 we describe in detail one of the consistency protocols outlined in the previous section. Section 5 discusses how failures can be managed in all consistency protocols, since the failure handling procedures are common for all of them. Section 6 describes the actions taken when a faulty node recovers, and finally, in section 7 we provide some concluding remarks.

2 Common Concepts

Within Globdata, each replica of the database communicates with the other replicas through the local consistency managers. Consistency managers are, thus, in charge of implementing the consistency protocols which drive a Globdata system. Globdata's proposed architecture places them as mediators for every data access action the local sessions (i.e., transactions) perform.

Some of the protocols use lazy replication. Consequently, not all system nodes have the latest version of each object. As a result, considering a particular object, the set of nodes can be assigned one of the following roles:

- *Owner node*: It is the node where the object has been created. It is the manager for *access confirmation requests* (see below) for the object; i.e., it allows or denies these requests when the session initiators ask it about them.
- *Synchronous nodes*: They are the nodes preconfigured in the system to maintain the up-to-date replicas of the object. Their number for each object is preconfigured. They are needed for fault tolerance.
- *Deferred nodes*: These nodes do not usually maintain up-to-date replicas of the object, although they may have synchronous replicas sometimes (at least, when the session that has caused the latest object version has been initiated in one of them).

The approach we take to concurrency control is influenced by the fact that data may be lazily replicated, voiding the possibility of using traditional locks. Each object is, thus, assigned an owner node which controls accesses to that object. This control is only enforced when sessions try to commit.

When a session initiates its commit phase, the owner nodes of the accessed objects are contacted to discover if the session has used the latest committed versions of the objects. In this case, the owners validate the accesses, and the session can be committed. If any of the objects read by the session is reported to have an outdated version, the session is flagged for abortion. We refer to these contact actions as "*access confirmation requests*", since they only ask the owners about the correctness of the object versions used by the session.

Globdata has been designed for distributed systems that may use wide area networks. Within WANs, nodes may fail and network partitions may happen. Globdata needs, thus, the services of an appropriate membership protocol. This membership service assigns static node identifiers to the nodes that form the system, reducing the length of a node identifier. Consistency protocols require node identifiers to record the object ownership, as well as the role assigned to each node when a particular object is considered.

Since we use *access confirmation request* management, session identifiers (or SID's, for short) must include in their fields the identifier of the node that has initiated them. Thus, in case of node failure, the consistency protocol knows which granted access requests belonged to the faulty node, being able to manage the situation appropriately.

Objects also use an identification structure similar to sessions. For instance, object identifiers (OIDs) hold the static identifier of the node that has created them. When a node fails, all its objects are inherited by a new manager. However, this change in the management has to be written down by all live nodes in some records that they hold. These tables are later used to find the manager for each object.

Moreover, we also need to hold an object version associated to each object. This helps when a node recovers, since it only needs to send to one of its neighbors the last object version known for each object maintained in the recovering database. When a node receives that message, it replies with all changes that must be applied to bring the recovering database up to date.

An object version number includes the SID of the session that has written its last value. This information is used to build the graph of causal precedent sessions in the consistency protocols based on session updates (see sections 3.3 and 4 for details).

In case of partition failures; i.e., when some network links fail and two or more node subgroups[1] remain isolated, we only allow the subgroup with a majority of up-to-date replicas of a given object to work with sessions where that object is

[1] We use the term "*subgroup*" for naming each of the sets of communicating nodes that have appeared after the network partition. Each subgroup is unable to communicate with any other.

involved. If a partition arises or if some node fails, the role of the faulty owners[2] is moved to one or more than one of the remaining live managers.

3 Overall Protocol Descriptions

All consistency protocols follow a common basic algorithm that consists of the following steps:

1. Each consistency manager maintains in the database a set of tables with meta-data, that is, data needed to implement a particular consistency protocol. In particular, all the protocols keep in meta-data tables information about the versions of the objects stored in the local database.
   When a local session tries to read an object in a database access, the local consistency manager checks its version in the local database. If the consistency manager finds that the local version is outdated, it sends a message to the owner node of the object requesting the latest updates. The local session is blocked until these updates have arrived and have been applied to the local database. In fact, the query that originated this read access still has not been executed in this database.
   Write accesses are not managed this way. They are directly applied to the database without any check. However, although no check is made some records are taken for all accesses. Actually, for each access made in a session the sets of objects read or written in those accesses are written in the meta-data tables to be able to build later the readset and writeset of each session.
2. When a session tries to commit, its local consistency manager retrieves the read and write sets of the session and sends an *access confirmation request* to each one of the owners of the objects involved in that session, including for each object, its OID, object version and access mode (read or write).
   When an owner receives one of the above messages, it compares the version of the object within the message with the latest versions for that object. If they are equal, the owner grants the requested access permission and sets this SID as the current owner of that access grant. The access grant is exclusive if the access mode was "write" or shared if the access mode was "read". The grant is assigned to the requester until its session has propagated its updates to, at least, the synchronous set of replicas for the objects involved in the session. Once the session has been committed in this set of nodes, the session initiator releases the access grants after changing the version of their managed objects accessed in write mode within the requesting session.
   While a session holds access grants, other sessions that request the same grants in conflicting modes receive a deny reply to their requests. When such a reply is delivered, the consistency manager that requested that grant aborts the corresponding session.

[2] And this only can happen if we ensure that at least an up-to-date replica of the managed object remains in the live group.

Using the above approach, different consistency protocol variants can be designed providing support for different kinds of consistency modes. Next we describe the three consistency modes we care about within Globdata. Later we show how different alternatives in the overall approach lead to those modes, and finally we outline the protocols resulting from such alternatives.

3.1 Consistency Modes

Three different consistency modes are considered in Globdata. Each session that uses the Globdata services may only use one of these modes at a given time, although there are methods in the API for changing the consistency mode associated to the application sessions. These consistency modes are:

- *Plain*: This mode only allows isolated read requests and guarantees that all accessed objects respect session causal commit order. However, the versions of the accessed objects may not be the latest ones; i.e., they may be out of date.
- *Checkout*: This mode is a permissive variation of the transaction mode discussed below. In checkout mode, the isolation property is not guaranteed. Thus, if several sessions have read accesses on a given object, one of these sessions is allowed to promote its access to write mode (this can break serializability, thus, within a standard transactional setting this would lead to the blocking of the promoting transaction, or to the abort of a transaction). However if two of these sessions promote their read accesses to write mode for the same object, one of them is aborted.
- *Transaction*: The usual transaction guarantees (basically, the ACID properties) must be provided.

We have described the consistency modes in increasing order of restrictiveness; i.e., transaction mode allows less concurrency conflicts than checkout mode, and checkout mode less than plain mode. So, when two sessions that use different consistency modes conflict, some rules have to be adopted to resolve that conflict. In practice, we use the rules of the more restrictive mode.

3.2 Protocol Alternatives

The consistency algorithm shown at the beginning of section 3 admits various choices in several of its steps. We outline these options and their multiple alternatives now. The chosen characteristics generate different kinds of consistency protocols that we describe later.

These are the algorithm characteristics that admit multiple choices:

- *Update multicast when a session commits*: A protocol may broadcast the session changes when it commits. In this case, all object replicas are synchronous and if the commit procedure is careful with the session commit order, all consistency modes can be easily guaranteed.

On the other hand, the session changes can be multicast only to a reduced set of replicas (those placed in the preconfigured synchronous replicas for the objects involved in the session). This management leads to lazy replication. Since different objects may reside in different sets of synchronous nodes, special care has to be taken to guarantee all consistency modes when lazy replication is used.

- *Per object or per session update propagation*: If lazy replication is used, when a session commits we have to choose whether all session updates have to be propagated to all nodes where at least a synchronous replica of any updated object exists, or the updates are propagated per object. The second option implies that if two objects have been modified in a session and they do not reside in the same set of synchronous nodes, then we only propagate to each set of synchronous replicas the changes that involve one object, but not all session changes.
  The first option guarantees that plain mode consistency is satisfied, however the second option does not always guarantee that.

Additionally, depending on the consistency mode that a session uses, the actions that a consistency manager has to do when a session reads objects or commits are the following:

- *On read accesses*: If plain mode is being used, the object version stored in the local database need not be the latest one. However, causal commit consistency must be guaranteed, so the session changes have to be stored in a database only when all sessions that precede the one that is committing have been already stored in that database. So, plain mode is not problematic on read accesses if some care has been taken when commit operations were made using other consistency modes.
  For checkout and transaction modes, when a read access is requested, the consistency manager has to ensure that the latest object version exists in the local database to avoid the session being rolled back later. If this version is not present (and this may only happen when lazy replication is used), it has to be requested to the object owner, notifying it about the object version stored in the database of that session initiator. When this request is received by the owner, the requested version has to be returned to the session initiator. However, since plain mode must be usually guaranteed, not only the latest version is needed, but all object versions between that stored in the initiator database and the latest one. Moreover, not only these objects must be returned: the initiator also needs all other session changes for all the sessions that have caused these object updates.
- *On commit time*: There are no commit operations in plain consistency mode because only read operations are allowed in that mode.
  In checkout mode, the read accesses are treated the same way as in transaction mode, but the commit procedure differs a bit. In this mode, if a session has read an object version that at commit time is not the latest one, it is not aborted. However, objects in the write set have to be checked at commit

time and their version must be equal to the latest one. If this does not happen, the session is aborted. To sum up, read objects may change while the session is executing, but written objects must not be overwritten by other concurrent checkout sessions.
In transaction mode, both read and written objects must have their latest version when the session is committed. If a subsequent change has been made by another concurrent session, the one that is terminating must be aborted.

3.3 Consistency Protocols

We present three consistency protocols that have different characteristics. They are the following:

- *Full object broadcasting*: This protocol uses immediate updates in all system databases, so it does not use lazy replication. Thus, the writeset of a committed session is broadcast to all system nodes and it is immediately applied. Of course, not all sessions are committed, since object owners must grant the access confirmation to do so. These access permissions depend on the consistency mode used by the session.
  It supports the three consistency modes and read accesses do not need any additional action (they can be locally performed without any special handling).
- *Simple object update*: This protocol uses lazy replication and object updates, instead of session updates. As a result, although this protocol complies with all consistency modes, plain mode requires more effort, since the way sessions that use transaction or checkout mode propagate their updates does not provide the guarantees needed by plain mode accesses.
  Note that at commit time, the updates are only propagated to the preconfigured synchronous replicas of each modified object. Thus, it is possible that the full effects of a session are not reflected at all nodes that have received an update message from it: a node can have a synchronous replica for one of the involved objects, and a deferred replica for one of the other objects.
  When a read operation needs a more recent version than the one stored in the local database, only the latest version is requested (and obtained) from the object owner. No other contents need to be transferred.
- *Session set update*: This protocol uses lazy replication and session updates; i.e., when the updates are transferred to other nodes, not only the object changes are transmitted to their synchronous replicas, but all session updates (i.e., the session writeset) are transferred to all nodes that have at least a synchronous replica of one of the changed objects.
  To support plain mode, an additional problem appears: before the effects of a session can be applied, all sessions preceding it in causal order need to have been applied before to the same database. Sometimes, however, this has not yet done. For instance, when an object has a deferred replica in a given node that has not received any update for a long period of time and

some of the objects that maintain a synchronous replica in that node have been modified in the same session.
The sequence of steps needed to get a group of missing sessions is the following:

1. A request is sent to the object owner, asking for the session that has made the latest change on that object and all its precedent sessions using causal commit order. All nodes maintain a log of committed sessions, until they have been applied to all system nodes (when this happens, the session is removed from the log).
   The request also carries the local (and out-of-date) object version number for the requested object. So, the object owner, following the SID's stored in the object versions and scanning the logs, is able to build the graph of precedent sessions.
2. The object owner replies with the graph of sessions. This graph has as its root the session that has caused the latest update on the requested object and that also includes all its precedent sessions following causal commit order. This causal commit order is easy to find using the SID's stored in the object versions. For each session, the log also maintains its writeset and readset. So, with this information, all the graph can be built. To add more layers to the graph, only the readsets of the current leaves of the graph have to be inspected, and all the sessions that appear in the object versions of those readset objects are included in that layer. When a session does not appear in the log, it can not be added to the graph since this means that all its changes have been applied in all system nodes.
   The graph built in this step only maintains the SID's of the sessions, not their readsets or writesets.
3. The requester checks the received graph and scans it in depth order, starting at the leaves and removing from the graph all sessions that have already been applied to the local database. When the scanning arrives to a level where no session has been removed, this procedure terminates. The resulting graph is returned to the object owner node.
4. The object owner receives the returned graph and replies with the readsets and writesets of all the sessions found in the graph. These data is stored in the requester node when it is received, terminating thus the retrieval of the precedent sessions.

4 Algorithm Specification

This section provides additional details for one of the proposed consistency protocol. In this protocol, the message transport is assumed reliable (in the sense of TCP/IP reliability) and a membership service exists, which notifies the system nodes about the failures and recoveries of other system nodes.

4.1 Session Set Update Protocol

This protocol transfers the whole set of session updates each time a session is committed. This set of updates is sent to all nodes that have at least a synchronous replica for one of the objects updated in that session. As a result, plain mode can be easily supported, without needing any special action; i.e., the reads can be locally completed without needing any further message exchange.

Each consistency manager executes the following algorithm:

- Every node maintains a log containing every session applied in its local database. A process is run asynchronously in every node in order to update each node in the system. When this asynchronous process can ensure that a session has been applied in every node, this fact is indicated and the session can be eliminated from the logs.
- When a node detects an out-of-date object in a read request, it locates the owner node of the object (N_o), and sends a request message to it in order to update its object copy. This request message contains the identifier and version of the out-of-date object.
- The owner node receives the request message, and looks at its meta-data for the set of causal dependent sessions, needed to update the requested object from the given version to the version held in the local database of the owner node. This process is performed by the following algorithm:
  - The owner node looks at the meta-data for the last session that modified the requested object (T_o). In this session, other objects have been read (the readset of T_o or $R(T_o)$).
  - For each object o_i contained in $R(T_o)$, the node should search its log for every session T_j having o_i in its writeset (T_j causally precedes T_o).
  - The node takes then T_o and every T_j and composes a graph representing the causal dependencies of T_o and every T_j.
  - For the last sessions added to the composed graph, the algorithm iterates in order to include every causal dependency in the causal graph. The iteration ends when all the logged sessions with causal precedence have been included in the causal graph.
- The composed graph (that is actually containing statements of sessions) is sent to the requesting node, which analyzes the graph in order to eliminate already applied sessions, and to determine whether each session in the graph can be applied in its local database.
  The graph can be cut out at a session T when this session T has been already applied in the requesting node. This occurs when every object in the writeset of T has a lower version than the version held in the local database.
  In addition, a session in the graph cannot be applied to the local database when an object contained in its readset has a higher version than the version held in the local database (that is, there exists causal precedent session yet unknown for the requesting node), and this out-of-date object must be updated before.

- When the cutting out process is completed, the requesting node sends a new message to the owner node, requesting the complete writeset (values and versions) of the meta-session resulting from compacting the graph.
- The owner replies with a message that holds the writesets of each session included in the previous request. This information can be extracted from the meta-data tables, since all these sessions have been locally applied by this owner node and it knows about all these writesets.

Note that "plain" consistency mode is directly implemented, granting that each update preserves the causal consistency of the local view of the database.

The amount of information needed to provide this functionality consists of:

- A log of every applied session for each node. This implies redundancy in the logs.
- A session is kept in the log until an asynchronous process determines that the session has been applied in every node.

5 Failure Analysis

Several failure scenarios must be considered to ensure that these protocols work when failures arise. We consider two kinds of situations here. The first one deals with the completion of the steps given in the protocols specification. The second one deals with the migration of the managing role between nodes. We discuss both of them next.

5.1 Algorithm Completion

When a node fails, our membership monitor detects this failure and notifies all of the remaining nodes about this event. Partition failures are notified the same way, but in this second case the set of faulty nodes may be bigger.

Let us see what happens with the sessions initiated by a faulty node. We can distinguish the following cases, according to the step at which the failure happens:

- If one of these sessions has not surpassed its access confirmation granting step of the algorithm, no record of that session exists in any of the live nodes. So, that session can be discarded. In fact, when its host node recovers, it must abort that session, forcing its application to repeat the work.
- If the session fails once it has obtained the remote access grants, but before it has multicast any update a similar situation arises. No record of the session updates can be found in any of the live nodes, so the session cannot be terminated in the remaining nodes. As a result, that session must be aborted when its host node recovers. However, the faulty node has obtained some access grants and this may prevent other sessions to work.
  To avoid this, a solution is provided. Since the multicasts are atomic and reliable, if an object update has been received by one of its synchronous

replicas, all of them have received this update. So, when the membership service notifies the failure of a node, all object owners scan their grants lists. If some access has been granted to a session initiated in the faulty node, the access granter (that is also the owner of the object protected by that grant) has to check if some update multicast has been received associated to the SID that requested that access. If no such an update was received, the grant can be released, otherwise the following point has to be considered.

- If the session has at least initiated the update multicast, its updates may have arrived to other nodes. In this case, we need to perform the same actions as in the previous case. The grants held by this session have to be released. Since the updates have been received, and this may only arise if all grants were obtained and all changes made (but still not committed) in the original node, no additional access grant will be needed. Since the session initiator node has failed, the grants are not needed by any other node that replaces the faulty one, because this hypothetical replacer node already has committed this session. As a result, the access grants have already been used correctly and they must be released now.

  No other case needs consideration. Perhaps the session had not been completed yet, but this only means that it held some grants that have been released as a consequence of the steps explained above. So, the session has been completed now.
- If the node has failed once the update multicast was initiated but before the "updatever" message[3] has been sent (assuming that this kind of message is needed for that session), then another problem arises because the node that eventually inherits the object ownership does not have the version number information needed to send that message.

  When the node promoted as object owner starts its new role, some actions must be taken. It already knows that it maintains a synchronous replica. It may decide that another node (or more than one node) has to be promoted to hold a synchronous replica. This fact depends on the type of configuration needed. Moreover, it has to broadcast a message to all accessible nodes with deferred replicas of objects whose management it has inherited. This message contains the version number of these inherited objects. The deferred nodes will reply this message indicating the promoted node whether their current replica is actually out-of-date or not.

5.2 Role Migration

When a node fails, all the objects it managed have to be managed by one of the live nodes. As a result, the object ownership initially managed in that node has to be migrated to one of the other nodes. So, we have to discuss two tasks in this section. The first one deals with the criterion followed to elect the node

[3] This message is needed to notify all deferred replicas that hold the latest version number of the object until now, that this version has changed and that they do not have an up-to-date version in their local database.

that will replace the faulty one. The second task deals with the migration of the access granting management.

Let us see how these tasks may be carried out:

- The election of the replacer node is based on the static identifiers associated to all preconfigured members of the system. Since each node has an identifier of this kind, we only need to choose the live node with a synchronous replica that has the lowest identifier among all those that are greater than that of the faulty node (or the lowest one, if the faulty node had the greatest identifier). To be able to elect the new manager, a majority of synchronous replicas must still exist in the system.
  In case of using an even number of synchronous replicas, some criterion is needed to break the tie in case of network partitions. For instance, the subgroup that holds the node with lowest identifier among the previous set of synchronous replicas will maintain the new manager.
  We assume that the number of synchronous replicas is known in advance. In case of network partitions, it may arise that the owner for a given object remains alive, but the greater part of its synchronous replicas remain unavailable. If such a situation arises, the current manager must give up its role. A problem appears here. If the majority of synchronous replicas are in another subgroup after the network partition, one of them will take up the management role, according to the procedure described in the previous paragraph. However, if those replicas have failed, no other session in the whole system will be able to use that object again.
- The node that inherits the object ownership has to ask all the others about the access grants they have; i.e., it has to know which of the grants it manages has an owner and who is that owner. To this end, each node that holds a grant that was managed by the faulty node, sends an ownership message to the new manager. This message contains the identifier of the object associated to the grant, the access mode, and the SID of the session that holds it. If a node has no grants, it sends an empty message.
  A timeout is set by the new manager to receive all these messages. If some message has not been received in this period, an explicit request is sent to that node.
  Note that the criterion used to select the replacer node is known by all managers, so no message is needed to make public the identity of the new manager.
  In case of network partitions, this is also true. If one subgroup loses its manager for a given object, no session that accesses that object will be allowed in that subgroup. Moreover, all grants maintained in that subgroup have to be released, and all sessions that had those access grants must be aborted.

6 Recovery Analysis

When a node recovers, the membership monitor notifies all live nodes about it. So, during the reconfiguration of the system state, two tasks must be performed

related to the new node: recovery of the object ownership (for all the objects initially created by that node, if any) and updates of its local database to make it consistent with those of the other nodes.

In order to recover the object ownership no special action must be taken by the new node. One of the previously active nodes had inherited the object management for all objects of the currently recovered node. This node must send to the original owner all its information regarding access grants. Until this message is received by the recovered node, it can't manage its incoming access grant requests or releases. It has to hold them temporarily. Once the message and the object ownership have been transferred to the recovered node, it resumes the access grant management. It may happen that an access request or release arrives later to the node that has returned the object ownership to the recovered node. If that happens, all these messages must be forwarded to the current manager. No special action is needed by the sender of such messages. The forwarding is done by the message receiver, who knows which is the current manager for the object involved in that request or release.

Another change is needed if the recovered process belongs to the class of nodes that must maintain synchronous replicas of several objects. The owners of those objects have to reinclude it in the set of synchronous replicas, and possibly one of those replicas has to be degraded to the deferred category (although it initially maintains an up-to-date copy, but eventually it will become obsolete if it is degraded). No message exchange is needed to do so.

Note also that all the tasks explained above are made during the reconfiguration steps. In these steps no new sessions are allowed and the session management is temporarily disabled.

In this recovery procedure, the new node sends to one of its neighbors (for instance, the one that has the greatest probability of having inherited its object ownership, i.e., the one that has an identifier immediately greater than its) a request of the updates. In this request it includes the OIDs and object versions that it has in its local database. As a reply, its neighbor will send a message with all updates that have to be made to make consistent its database to those of the other nodes.

7 Concluding Remarks

Global data access is increasingly important to a large number of Internet-based applications. Such access has to be provided with guarantees of data availability and consistency. This work proposes an approach to build such applications more easily and reliably by means of replicating commercial grade, reliable database engines, and running adequate protocols to keep their data consistent.

Concurrency control in these protocols is quite optimistic, since access confirmation requests may be made when local updates have concluded. This behavior is suitable for distributed replicated databases whose applications usually work only with "local" data, i.e., data that has been created by other local applica-

tions, but where replication is needed to improve the access over "remote" data when the greater part of the sessions use "read only" access.

The algorithms discussed are being implemented within the Globdata Project, where they will be accesed by different kinds of applications, with different access patterns, which will give us in the future, the oportunity to verify the best performing approach for each kind of application.

References

1. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.
2. Fabrizio Ferrandina, Thorsten Meyer, and Roberto Zicari. Correctness of lazy database updates for object database systems. In *POS*, pages 284–301, 1994.
3. Fabrizio Ferrandina, Thorsten Meyer, and Roberto Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 261–272, Santiago, Chile, 1994.
4. Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. pages 173–182, 1996.
5. Maurice Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.
6. Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Sys.*, 15(1):96–124, March 1990.
7. ITI, UPNA, and FFCUL. COPLA programming interface, deliverable 04 (workpackage 01). Technical report, Globdata, IST Programme, project number: IST-1999-20997, June 2001.
8. Sushil Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. on Database Sys.*, 15(2):230–280, June 1990.
9. Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *International Conference on Distributed Computing Systems*, pages 156–163, 1998.
10. Narayanan Krishnakumar and Arthur J. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Trans. on Database Sys.*, 19(4):586–625, December 1994.
11. Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. on Comp. Sys.*, 10(4):360–391, November 1992.
12. Erhard Rahm. Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems. *ACM Trans. on Database Sys.*, 18(2):333–377, June 1993.
13. Ouri Wolfson and Sushil Jajodia. Distributed algorithms for dynamic replication of data. pages 149–163, 1992.

μCRL specification of event notification in JavaSpaces™*

Jaco van de Pol and Miguel Valero Espada

Centrum voor Wiskunde en Informatica,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{Jaco.van.de.Pol, Miguel.Valero.Espada}@cwi.nl

Abstract. In this paper, we extend the formal specification of the JavaSpaces architecture presented in [18] with the event notification mechanism. Processes running on a JavaSpaces system can register their interest in incoming entries. The space informs the arrival of matching entries by sending events to the registered processes. We use μCRL, a language that combines abstract data types with process algebra, to model a formal abstraction of this mechanism. The purpose of this work, in combination with the previous one, is to verify properties of the JavaSpaces technology and to allow automatic model checking of distributed applications built under it.

1 Introduction

The parallel composition of simple behavior agents can produce complicated systems. Distributed applications have to manage with the communication and synchronization between processes across heterogeneous networks, dealing with latencies, partial failures and system incompatibilities. Hence coordination architectures attempt to assist programmers at the difficult task of designing and implementing reliable distributed systems.

JavaSpaces™ [16] technology is a Sun Microsystems, Inc. coordination architecture, implemented as a Jini™ [17] service. It gives support to two programming styles of processes coordination: the shared dataspace (Linda [8] like style) and a reactive style. External agents communicate by sharing objects through the space, by means of some basic primitives. They can basically write and look up objects but they can also express their interest in incoming entries, by registering using the *notify* primitive. Then the space is charged to inform the agents the presence of suitable entries by sending events. The external processes "react" to the arrival of new entries in the space.

In a previous paper [18], we studied the basic features of the shared dataspace style, now we are going to present the formal specification of the notification mechanism using μCRL [13, 11], a language which merges the standard process

* Partially supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009.

algebra ACP [1] and abstract data types. By extending the model with the new operation, we allow to prototype and verify more JavaSpaces applications. The verification of the system is done by using the combination of the μCRL tool set [2] and the Cæsar Aldébaran Development Package (CADP) [10].

During the implementation of the μCRL model we had to face several difficulties in interpretation of the JavaSpaces specification. There are some issues that JavaSpaces specification leaves unclear or ambiguous and that are actually solved in the implementation. In our work, we attempt to clarify and resolve this lack of precision and detail.

This paper is structured as follows. After this introduction, we present the JavaSpaces specification and the μCRL language. We continue with the study of the formalism of the JavaSpaces architecture focusing on the notify primitive. Then, we illustrate the specification by modeling and model checking a simple application. The paper finishes with the conclusion and some references to other related works. The specification and some examples can be found at: "http://www.cwi.nl/~miguel/JavaSpaces/".

2 JavaSpaces

JavaSpaces is both an application program interface (API) and a distributed programming model. Agents can interact simultaneously with a shared dataspace of objects, the space handles the details of concurrent access to the data. Agents of applications are "loosely coupled", they do not communicate with each other directly but by sharing information via the common space. They use a small set of primitives described in Figure 1:

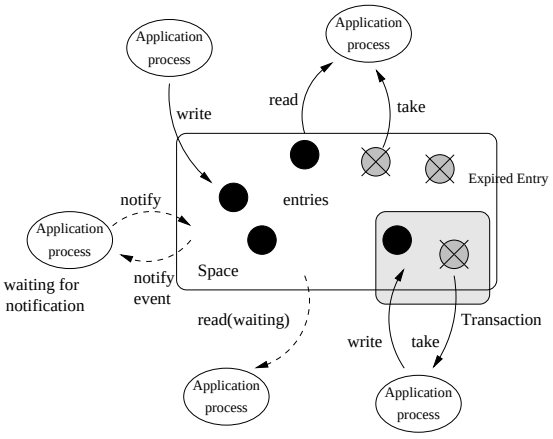


Fig. 1. JavaSpaces architecture overview

A *write* operation places a copy of an entry into the space. Entries can be located by "associative lookup" implemented by *templates*. Processes find the

entries they are interested in by expressing constraints about their contents without having any information about the object identification, owner or location. A *read* request returns a copy of an object from the space that matches the provided *template*, or *null* if no object has been found. If no matching entries are in the space, then *read* may wait a user-specified amount of time (*timeout*) until a matching entry arrives in the space. *ReadIfExists* performs exactly like *read*, but it only blocks if there are matching objects in the space but they have conflicting locks from one or more other transactions. *Take* and *takeIfExists* are the *destructive* versions of *read* and *readIfExists*: once an object has been returned, it is removed from the space.

The *notify* primitive is used to express interest in future incoming objects. The agent provides a template and the space will notify the agent when a matching object has arrived, by means of an event. Three entities are involved in the notification mechanism: The space is the source of events, it fires an event when an entry matches a registration. The destinations, called listeners, wait for the arrival of events and "react" to them. And the application process which register the listeners to be notified. The registration is done by the synchronous action *notify* based on the JavaSpaces specification [12].

```
public interface JavaSpace {
   ...;
   EventRegistration notify(Entry tmpl, Transaction txn,
         RemoteEventListener listener, long lease,
         MarshalledObject handback)
      throws RemoteException, TransactionException;
   ...;
}
```

The primitive gets as arguments the template to match entries, the reference to a transaction, the reference to the remote event listener, the lease and a handback used to pass information from the application process to the listeners. The space returns an *eventRegistration* object, which includes the registration identification number (the space assigns a identification number to any new registration), the granted lease and the initial sequence number for events generated from the *notify* registration. Every matching entry will increase by one the sequence number of the registrations. And newly generated event will contain a sequence number greater than the previous one.

JavaSpaces also provide support to leasing and transactions, from the Jini architecture [17]:

JavaSpaces supports a transactional model ensuring that a set of grouped operations are performed on the space atomically, in such a way that either all of them complete or none are executed. Transactions affect the behavior of the primitives, e.g. an object written within a transaction is not externally accessible until the transaction commits, the insertion will never be visible if the transactions aborts. Transactions provide a means for enforcing consistency. Transactions in JavaSpaces preserve the ACID properties: Atomicity, Consistency, Isolation and Durability.

JavaSpaces allocates resources for a fixed period of time, by associating a lease to the resource. The lease model is beneficial in distributed systems where partial failures can produce waste of resources. The space determines the time during which an object can be stored in the repository before being automatically removed. Also transactions are subject to leasing, an exception is sent when the lease of a transaction has been expired. Leases can always be renewed or canceled.

In the paper [18] we have presented the μCRL specification of the primitives: *write, read, take, takeIfExists and readIfExists*, leases and transactions. Now we are going to focus on the *notify* operation.

To know more about JavaSpaces, please consult the references [12, 16].

3 Introduction to μCRL

A μCRL specification is composed by two parts. First, the definition of the data types, called **sorts**. A sort consists of a signature in which a set of function symbols, and a list of axioms are declared. For example, the specification of the booleans (*Bool*) with the conjunction operator (*and*) is defined as follows:

```
sort  Bool
func  T,F:→Bool
map   and: Bool×Bool→Bool
var   b: Bool
rew   and(T, b) = b
      and(F, b) = F
```

The keyword **func** denotes the *constructor* function symbols and **map** is used to declare additional functions for a sort. We can add equations using variables (declared after **rew** and **var**) to specify the function symbols. The declaration of the sort *Bool* must be included in every μCRL specification because booleans are used for modeling the guards in the "if-then-else" construction.

The second part of the specification consists of the process definition. The basic expressions are actions and process variables. *Actions* represent events in the system, are declared using the keyword **act** followed by an action name and the sorts of data with which they are parameterized. Actions in μCRL are considered atomic. There are two predefined constants: δ which represents deadlock, and τ which is a hidden action. *Process variables* abbreviate processes, and are used for recursive specifications. *Process operators* define how the process terms are combined. We can use:

- The sequential, alternative and parallel composition $(.,+,\|)$ process operators.
- **sum** $(\sum)$ to express the possibility of infinite choice of one element of a sort.
- The conditional expression "if-then-else" denoted $p \lhd b \rhd q$, where b is a boolean expression, p and q process terms. If b is true then the system behaves as p otherwise it behaves like q.

They keyword **comm** specifies that two actions may synchronize. If two actions are able to synchronize we can force that they occur always in communication using the operator ∂_H. The operator τ_I hides enclosed actions by renaming into τ actions. The initial behavior of the system can be specified with the keyword **init** followed by a process term:

$\mathsf{System} = \tau_I \partial_H (p_0 \parallel p_1 \parallel ...)$
init System

4 μCRL Specification

The space is modeled as a single process called *javaspace*. External agents are implemented as separate processes executed in parallel with the space. A JavaSpaces system is specified in μCRL as follows:

$$\mathsf{System} = \tau_I \partial_H (javaspace(...) \parallel external\_P_0(id_0 : Nat, ...) \parallel external\_P_1(id_1 : Nat, ...) \parallel ...)$$

The arguments of the *javaspace* process represent the current state of the space. They are composed by: stored objects, active transactions, the current time, active operations, notify registrations, et cetera... External processes interact with the space by means of a set of synchronous actions, derived from the JavaSpaces API. Every process has a unique identification number used by the space to control the access to the common repository. Processes use the sort *Entry* to encapsulate the shared data. In the JavaSpaces specification, an entry corresponds to a serializable Java™ object which implements the public interface Entry (with some other restrictions). In our model, entries are represented by a **sort**. Users can define their own data structure according to the application requirements. The insertion of a new entry into the space is done with the action *write* which has four arguments: the process identification number of the sort *Nat* (naturals), the entry of the sort *Entry*, the lease of naturals and the reference to a transaction (*null* if it is not submitted to any one). When the space receives a write request, it automatically encapsulates the entry, with its lease and the reference to the transaction, in an new data sort (*Object*) and stores it in the database which has the structure of a *Set*.

Look up primitives could be classified as: *destructive* and *non-destructive*, depending on whether the item is removed or not after the execution of the action, and in *blocking* and *non-blocking* depending on whether the process waits until it receives the requested item. We can invoke destructive look ups (*take*) or non-destructive (*read*), setting up the time during which the action blocks.

The JavaSpaces specification says that a look up request searches in the space for an Entry that matches the template provided in the action. If the match is found, a reference to a copy of the matching entry is returned. If no match is found, *null* is returned. We do not use templates to model the matching operation but by adding to every invocation one predicate, as argument, which determines if an Entry matches or not the action. This predicate belongs to the

sort *Query*, defined by the user according to the specification of the *Entry*. The sort must include the operator *test* used to perform the matching. An entry of the space will match a look up action if it satisfies the associated *test* predicate. The look up operations are not atomic. They are done by two synchronous actions; first the process makes the request and blocks waiting for an entry or for the timeout expiration. First, the space stores this request in a set with other pending requests and afterward the space returns a matching entry or the null value.

The behavior of all the primitives would be slightly different depending on whether they are executed under a transaction or not. Before focusing on the *notify* primitive let's see a small example of code illustrating the presented operations. The example is a recursive process which renames entries of type A to type B. It performs the operation under a transaction leased for one time unit. If the timeout of the transaction expires the space raises an exception and sent it to the process by means of a synchronous action (*exception*), then the process deadlocks:

proc ren(id:Nat) =
$\cdot\sum_{trc:Nat}$ (create(id, trc, S(0))
.(take(id, trc, FOREVER, isTypeA) + Exception(id, trc).δ)
.$\sum_{e:Entry}$ ((Return(id,e) + Exception(id, trc).δ)
.(write(id, renameToB(e), trc, FOREVER)+ Exception(id, trc).δ))
.(commit(id, trc) + Exception(id, trc).δ))
.ren(id).δ

Now, we are going to focus on the specification of the notify mechanism introduced in Section 2.

For simplicity, we have abstracted away the lease, the transaction and the handback but we will comment the inclusion of the first two fields later in this section. The template is replaced by a query. We assume the registration is done atomically, thus no events can be fired between the begin of the registration and its return. Therefore the initial sequence number of events will be zero. Due to these abstractions, the space only returns a single value representing the registration identification number. This operation is performed reliably so it cannot throw any exception. The action signature is:

sort Nat, Query
act notify: `Nat×Nat×Query×Nat`

The arguments are: the process identification number, the listener identification number, the query and the event registration identification number (provided by the space).

When the space synchronizes with a *notify* action, it stores the registration in a set. For each newly written entry it will check every registration to know whether it has to be notified or not. In other words, the space marks the registrations whose query matches the new entry. It also increments by one the event sequence number. The specification says that the space makes a "best effort"

to deliver the notifications, a notification event will be *eventually* sent to the registered listeners. The space does not guarantee the generation of an event for every matching entry, so several matching entries can be stored before the space decides to fire a message. The sequence numbers are useful to keep track of the events, as we will see on a small example in Section 5.

An event listener is an object that reacts to the reception of an event and that may be running remotely. The listener has a method (*notify*[1]) invoked whenever it receives a notification event. According to the JavaSpaces specification the *notify* call is synchronous so the space waits on a listener until the call finishes, but the JavaSpaces implementations are multi-threaded hence many different notify calls can be done concurrently. We modeled the *notify* operation with our single *javaspace* process, assuming that we have an implementation with enough threads to manage all the notifications of the system. The μCRL space delivers the event and doesn't wait until the end of the method call of the listener. This policy will not be admissible if there are too many listener registrations or if the *notify* methods are very slow (block the space for long periods) or never return. Our model would help programmers to take care about preserving the desirable behavior of the system, for example we will show one application in Section 5 the detection of a problem in a listener which arrives to a non desired blocked state.

The event sent by the space contains some data values. It includes the registration identification number, to allow a listener to distinguish the event as belonging to a particular registration and the sequence number of events, which can be used by listeners to know the number of events occurred from last notification.

In our model, listeners are going to be modeled as separate processes. μCRL does not allow the instantiation of processes on running time so listeners have to be defined at the beginning, according to the needs of the application. A listener has the following structure:

proc listener(id:Nat, d_0:D_0,...,d_n:D_n) =
$\sum_{registrationID:Nat}(\sum_{seq:Nat}$ (_Notify(id, registrationID, seq)
.*do_work*
.listener(id)))

Where d_0:D_0,...,d_n:D_n are the user defined arguments, and *_Notify* the action for receiving the event. The *.do_work* operation may be composed of any computation or any communication with other processes or with the space.

Messages travel over the network from the event source (the space) to the event destination (the listener); they are not delivered instantaneously nor reliably. Hence events may be lost and never reach their destinations. They may also arrive unordered because events can follow different paths. The event source can always duplicate messages because it cannot be sure whether the delivered events have arrived or not. To model this non-deterministic behavior, we specify

[1] Do not confuse with the registration notify method.

a separate process which represents the network situated between the source and the destination. This process stores the events in a fifo list, and can always:

- Receive: The network process receives and event from the space by means of the synchronous action $\_Notify$.
- Loose: It removes the first object of the list.
- Swap: It changes the order of the first two events that have the same destination.
- Duplicate: It replicates the first element of the list.
- Deliver: The networks dispenses an event to a listener using the action $\_notify$.

We parameterized the network process with a field counting the number of errors (looses, duplications and swaps). To keep finite the system we only allow a maximum number of errors. A reliable network would have a maximum number of errors equal to zero.

The complete system is composed of the parallel composition of the application processes, listeners, the space and the network. Figure 2 illustrates the model.

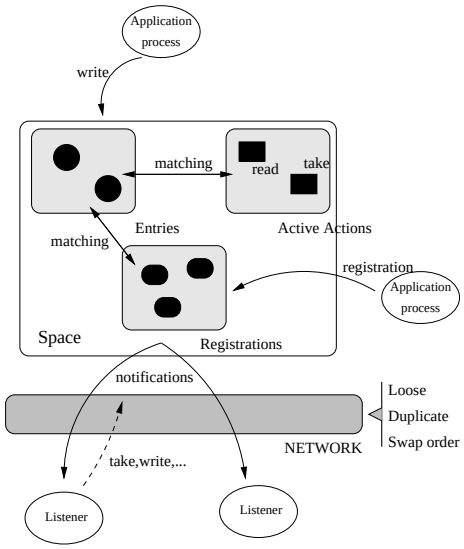


Fig. 2. Notification architecture

We can also add leasing to the registration mechanism. We proceed in the same way as for the look up primitives. The application process passes the requested lease to the space, it includes this value in a data field of the registration object. The space manages a centralized clock implemented as a discrete counter. The *javaspace* process increments this clock arbitrarily. Using this counter we can manage the expiration of the leases. When the registration lease expires the space automatically removes the object from the data base. Listeners can receive

events even if the registration has been removed, because the messages may be delayed on the network.

Notify can also be joined to a transaction. The space will send events when a matching entry is written under the same transaction of the registration or under the *null* transaction. If a transaction expires the joined registrations will be removed. We have not implemented these two issues (transactions and leasing) for the *notify* primitive, but according to its specification for the other primitives we foresee no major difficulties to do it. This concludes the presentation of the μCRL model, now let's analyze a simple application.

5 Example

In this section we are going to illustrate the use of the proposed specification to model check JavaSpaces applications by analyzing a simple example. The system consists of a process which registers a listener, expressing interest in any new data of the type *message*. When the listener receives the event, it just takes the message and prints *"Hello World"* (see the example in Chapter 8 of "JavaSpaces Principles, Patterns, and Practice" [12]). First we specify the sort *Entry*, which only has two possible values the compulsory null entry, and the message (we don't care about its content). The μCRL code is as follows:

```
sort  Entry
func  entryNull:→Entry
      message: →Entry
map   eq: Entry×Entry→Bool
var   e: Entry
rew   eq(entryNull,entryNull) = T
      eq(message, message) = T
      eq(entryNull, message) = F
      eq(message, entryNull) = F
```

We define two queries: *any* which matches any entry and *isMessage* which matches the entries of type *message*. Let's see the code:

```
sort  Query
func  any: →Query
      isMessage: →Query
map   test: Query×Entry→Bool
      eq: Query×Query→Bool
var   e: Entry
rew   test(any, e) = T
      test(isMessage, message) = T
      eq(any,any) = T
      eq(isMessage,isMessage) = T
      eq(any,isMessage) = F
      eq(isMessage,any) = F
```

The user application is composed by two processes. *Apps* executes the action *notify* registering the listener and the listener, that first gets the event and then tries to take the entry from the space. If the take is successful it does the action *HelloWorld*. Note that we have simplified the primitives *take* and *write* by removing the lease and the transaction. The code is:

proc apps(id: Nat, listenerID: Nat) =
$\sum_{registrationID:Nat}$ (notify(id, listenerID, isMessage, registrationID))
.write(id, token).δ

proc listener(id:Nat) =
$\sum_{registrationID:Nat}(\sum_{seq:Nat}$ (_Notify(id, registrationID, seq)))
.take(id, isMessage)
.waiting
.$\sum_{e:Entry}$ (Return(id,e))
.HelloWorld
.endNotify(id, registrationID, seq)
.listener(id)

Finally, the complete system is composed by the parallel composition of the space, the *apps* process, the listener and the network, which is allowed to commit one error.

System = $\partial_{\{write,Write,notify,Notify,\_notify,\_Notify,...\}}$
$(javaspace(0, emN, emA, 0)||Network(emE, 0, S(0))\,||$
$apps(0, S(0))\,||\,listener(S(0)))$

To each μCRL specification belongs a labeled transition system (LTS) being a directed graph, in which the nodes represent states and the edges are labeled with actions. If this transition system has a finite number of states the μCRL tool set can automatically generate this graph. Subsequently, the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP) can be used to visualise and to analyse this transition system. Figure 3 shows the generated LTS of the simple HelloWorld application where the action N corresponds to the communication between the *notify* action of the application process and the *Notify* action of the space, W to the write actions, $\_N$ corresponds to send an event from the space to the network, $\_\_N$ to deliver it from the network to the listener, T corresponds to a take request and Rt is the return of the *take*. *Duplicate, loose, HelloWorld, waiting* and *endNotify* are external actions informing about the execution of the system. Remark that the action *endNotify* is just a "printed" message, listeners do not synchronize with the space at the end of the *notify* invocation.

We can see in the figure 3 a desirable execution following the path: 0-1-2-3-4-7-9-11-13-15) and two undesired behaviors. The first is when the network looses the data (path: 0-1-2-3-5), thus the listener doesn't receive the message. The other is when the network duplicates the event(0-1-2-3-6-8-10-12-14-15-16-17-18-19-20), then listener tries to take two times the message. In this case the listener gets blocked *waiting* for a return that will never happen unless another

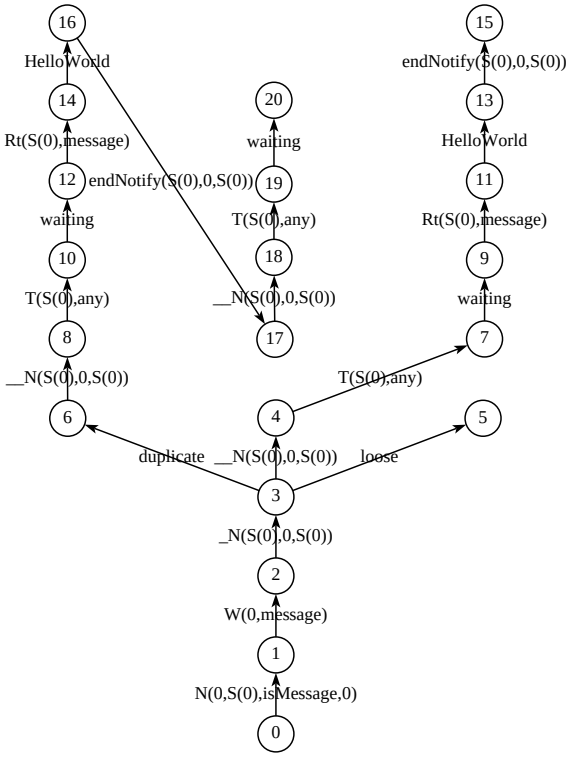


Fig. 3. LTS of HelloWorld in a non reliable network

process writes a new message. We can avoid the second undesired behavior by checking the event sequence number before trying to perform the take primitive. This is modeled with the following code:

proc listener(id:Nat, last:Nat) =
$\sum_{registrationID:Nat}(\sum_{seq:Nat}$ (_Notify(id, registrationID, seq)
.(take(id, isMessage)
.waiting
$\sum_{e:Entry}$ (Return(id,e))
.HelloWorld $\lhd$ gt(seq, last) $\rhd$ do_nothing)
.endNotify(id, registrationID, seq)
.listener(id)))

The listener only tries to take the message if the sequence number (*seq*) is greater than the sequence number of the last notification (*last*), otherwise it assumes that the event has been duplicated and finishes.

We can automatically verify some properties of the system using the Evaluator tool from the CADP package. These properties are expressed in temporal logic. We used the regular alternation-free μ-calculus formulas [15]. For example, the following formula means that every *notify* invocation of a listener finishes, in other words: after a *__N* there is always an *endNotify* with the same arguments:

```
[true*.'__N(.*)']mu X.(<true>true and [not 'endNotify(\(.*\))']X)
```

This formula does not hold for the first example of listener. The evaluator analyzes it and gives the counter example corresponding to already pointed path. However the second listener, which checks the sequence number of events satisfies the property.

6 Related Work

As we said, this work is an extension of the [18]. Our information of JavaSpaces is based upon the book [12], and the documentation from Sun on JavaSpaces [16] and Jini [17]. The latter document describes a.o. the concepts of leasing, transactions and distributed events. The basic ideas of JavaSpaces go back to the coordination language Linda [8].

Some work on the formalization of JavaSpaces (or other Linda-like languages) exist, notably [4, 5, 6, 7]. In these papers, an operational semantics of JavaSpaces programs is given by means of derivation rules. In fact, in this approach JavaSpaces programs become expressions in a special purpose process algebra. Those authors aim at general results, i.e. comparison with other coordination languages, expressiveness, and results on serializability of transactions. Verification of individual JavaSpaces programs wasn't aimed at.

Although we also take an operational approach, our technique is quite different. We model the Javaspace system, and the JavaSpaces programs as expressions in the well-known, general-purpose process algebra, μCRL [13]. This allows us to use the existing μCRL tool set [2] and the CADP tool set [10] for the verification of individual JavaSpaces programs. In our model, the JavaSpaces programs communicate with the JavaSpaces system synchronously.

Our technical approach is similar to the research in [9, 14]. In these papers, programs written under the Splice architecture [3] are verified. Both papers give an operational model of Splice in μCRL, and use the μCRL and CADP tool sets to analyse Splice programs. One of the main purposes of the Splice architecture is to have a fast data distribution of volatile data. To this end, the data storage is distributed, as opposed to the central storage in JavaSpaces. In Splice, data items are distributed by a publish/subscribe mechanism. Newer data items simply overwrite outdated items.

7 Conclusion

In this paper we studied the specification of the notify mechanism of the JavaSpaces architecture. We have found several difficulties in interpretation that we tried to solve. Some of these problems are solved in the implementation of JavaSpaces but not in its specification.

First, the specification says that an event can be duplicated by the event source. This issue is source of several questions: How and when does the space decide to send twice the same message? Has the *notify* call a timeout? Can listeners be notified during a notification? Other problem comes from the interpretation of "best effort"; the space will "eventually" send a event after a

write. But when does the space send a event? and when does it compress several matches in one notification?.

The *notify* call is "synchronous", so the space blocks until the end of the remote method. Which actions are listeners allowed to do in the *notify* method? What will happen if a listener never returns? What will be the difference between a single-threaded and a multi-threaded space?

We attempted to solve the unclear details by making assumptions about the behavior of the system. Our informations are not only based on the JavaSpaces specification, sometimes ambiguous, but also in the archives of the discussion group where some of these have been treated. See, for example:

`http://archives.java.sun.com/cgi-bin/wa?A2=ind9904&L=javaspaces-users&P=R3468&D=0&H=0&O=T&T=1`

`http://archives.java.sun.com/cgi-bin/wa?A2=ind0106&L=javaspaces-users&P=R2562&D=0&H=0&O=T&T=1`

The last part of the paper is dedicated to the study of a very simple JavaSpace applications. Although we cannot verify the correctness of the proposed model, we can see, in small examples, that the behavior corresponds to JavaSpaces specification. Together with the μCRL simulator this provides some validation of the model. We also present some ideas of how to verify properties of applications. In the same way we can study more complex problems.

The μCRL model of the notification mechanism may be used not only to model check JavaSpaces applications but also to study the architecture itself and resolve all kinds of unclear or ambiguous points.

References

[1] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[2] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. Langevelde, B. Lisser, and J.C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In *Proc. of CAV*, LNCS 2102, pages 250–254. Springer, 2001.

[3] M. Boasson. Control systems software. *IEEE Trans. on Automatic Control*, 38(7):1094–1106, July 1993.

[4] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proc. of SAC*, pages 146–155. ACM, 1999.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In *Proc. of AMAST*, LNCS 1816, pages 198–212. Springer, 2000.

[6] N. Busi and G. Zavattaro. On the serializability of transactions in JavaSpaces. In U. Montanari and V. Sassone, editors, *Electronic Notes in Theoretical Computer Science*, volume 54. Elsevier Science Publishers, 2001.

[7] Nadia Busi and Gianluigi Zavattaro. Publish/subscribe v.s. shared dataspace coordination infrastructures. In *Workshop on Web-based Infrastructures and Coordination Architectures for Collaborative Enterprises (WETICE-2001)*. IEEE Computer Society Press, 2001.

[8] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[9] P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In *Proc. of COORDINATION*, LNCS 1906, pages 335–340. Springer, 2000.

[10] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proc. of CAV*, LNCS 1102, pages 437–440. Springer, 1996.

[11] W. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer, 2000.

[12] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley, Reading, MA, USA, 1999.

[13] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra et al., editor, *Handbook of Process Algebra*, chapter 17. Elsevier, 2001.

[14] J.M.M. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings ACM SAC, Coordination Models, Languages and Applications*, page (to appear), Madrid, 2002. ACM press.

[15] R. Mateescu. *Verification des proprietes temporelles des programmes paralleles.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.

[16] SUN Microsystems. *JavaSpaces$^{\text{tm}}$ Service Specification*, 1.1 edition, October 2000. See http://java.sun.com/products/javaspaces/.

[17] SUN Microsystems. *Jini$^{\text{tm}}$ Technology Core Platform Specification*, 1.1 edition, October 2000. See http://www.sun.com/jini/specs/.

[18] J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces$^{\text{TM}}$ architecture using μcrl. In *Proc. of COORDINATION*, page (to appear). Springer, 2002.

Sistemas operativos para un receptor digital de televisión

Manuel Ramos Cabrer, Jorge García Duque, José Juan Pazos Arias, Cándido López García, and Alberto Gil Solla*

Departamento de Ingeniería Telemática,
Escuela Técnica Superior de Ingenieros de Telecomunicación,
Universidad de Vigo,
Campus de Lagoas – Marcosende, S/N,
36200 - Vigo (Pontevedra),
mramos@det.uvigo.es

Resumen En este artículo se presentan las características que debe presentar un sistema operativo para un receptor digital de televisión y se analizan dos posibilidades: la utilización de RT-Linux y un sistema operativo propio diseñado siguiendo la norma ITRON. En ambos casos se persigue que el sistema operativo proporcione los servicios necesarios al prototipo de receptor en que va a estar embebido y, al mismo tiempo, se adapte fácilmente a las necesidades de planificación conjunta de tareas de tiempo real críticas y no críticas que se presentan. Para ello se describe la experiencia adquirida con la implementación de un algoritmo de planificación en ambos sistemas. Por último, presentamos las conclusiones obtenidas.

1 Introducción

Desde la aparición del primer receptor de televisión en blanco y negro se han venido realizando constantes esfuerzos para mejorar la calidad de la imagen y del sonido. Una de las primeras mejoras significativas fue la introducción del color en las imágenes con diferentes tipos de sistemas de color: PAL (utilizado en España y en casi toda Europa), SECAM (utilizado en Francia) y NTSC (adoptado en EEUU).

En 1968 comenzó a investigarse en Japón sobre un sistema de televisión de alta definición. El resultado fue un sistema analógico denominado MUSE (MUltiple Sub-Nyquist Encoding), en el cual la transmisión era analógica previa compresión digital de la señal.

En EEUU se comienza a trabajar en un sistema de alta definición en 1987. En 1990, la Comisión Federal de Comunicaciones (FCC) recibió seis propuestas, de las cuales una era completamente digital. En 1993, la Comisión decide qué sistema adoptar, comenzando las emisiones un año después. A este acuerdo se le denominó Gran Alianza.

* Este trabajo ha sido financiado por el proyecto 1FD97-1195

Las grandes perspectivas de negocio de esta tecnología, hicieron que Europa impulsara el tema en el año 1986 con el nacimiento del proyecto EUREKA 95.

El objetivo del proyecto EUREKA 95 era desarrollar un sistema TV de alta definición (HDTV - High Definition TV) que empleara 1250 líneas, el doble del sistema PAL por razones de compatibilidad. Sin embargo, el sistema se concibió como una mejora de MAC (Multiplexed Analog Components). En 1986, la EBU (European Broadcasting Union) aprueba este formato para las emisiones vía satélite, pero pocas emisoras lo utilizan dado el mayor coste del receptor respecto al sistema PAL. Como ventaja, MAC nos ha dejado el sonido digital Nicam actual. En 1993, algunos fabricantes de receptores propusieron el sistema PAL Plus para introducir receptores con relación de aspecto 16:9 en lugar de los convencionales 4:3. Algunas emisoras utilizaron este estándar incentivadas por la Comisión Europea, al considerar este sistema un puente tecnológico hacia la alta definición.

Finalmente, en 1993 se celebra en Bonn una reunión de fabricantes y operadores con el objeto de coordinar todos los esfuerzos individuales e implicarse todos en un proyecto común. Este grupo se denominó DVB (Digital Video Broadcasting), y dio lugar a tres normas diferentes para la futura televisión europea para difusión terrestre (DVB-T), por cable (DVB-D) y por satélite (DVB-S).

La Televisión de Alta Definición (HDTV) reúne un conjunto de tecnologías avanzadas, que surgen de una cada vez mayor confluencia entre la televisión de entretenimiento y la electrónica digital. Estas nuevas tecnologías incluyen nuevos tipos de microprocesadores, pantallas de alta resolución y nuevas tecnologías de almacenamiento y transmisión de la información.

La televisión de alta definición promete ser el mayor avance en la tecnología de la imagen desde la aparición del color. La calidad de las señales de audio y vídeo es mucho mayor, y comparable ya con la que de un cine: en vídeo se doblan las resoluciones tanto horizontal como vertical, y en audio se incorpora el sonido digital. Además, la utilización de señales digitales abre la posibilidad de poder proporcionar una serie de servicios adicionales.

Entre las funciones más atractivas, desde el punto de vista de un usuario, que puede proporcionar esta tecnología están, por un lado, los servicios de comunicación, como pueden ser: acceso a servicios de comunicación de Internet: correo electrónico, WWW, etc; telecompra; servicios bancarios; recepción de mensajes comerciales; leer periódicos; etc.

Por otro lado, la utilización de un canal de retorno de datos permite la transformación del servicio de difusión de TV en lo que se ha venido a llamar Televisión Interactiva (ITV), que podrá proporcionar servicios tan atractivos como: sistema de pago por visión (Pay-per-view) o acceso condicional; juegos en red; recepción de vales descuento personalizados; visualizar retransmisiones eligiendo el ángulo de la cámara; responder a sondeos; etc.

El usuario podrá acceder a estas funciones a través de un simple mando a distancia que controla una interfaz gráfica de muy sencillo manejo.

Actualmente, el principal problema en la televisión digital es que solamente un receptor digital de televisión puede decodificar las señales digitales de audio

y vídeo, dado que será totalmente incompatible con los actuales estándares de televisión. Una posible solución es la utilización de un receptor especial entre la red de televisión digital y el televisor analógico. Este receptor, conocido como Set-Top Box (STB), puede decodificar señales digitales y transformarlas en señales analógicas válidas para los receptores analógicos actuales, y además proporciona el soporte necesario para la interactividad. Hasta la fecha, la mayoría de las redes se integraron verticalmente con un sólo proovedor de servicio controlando la cadena completa de distribución incluyendo cabeceras, sistemas de acceso condicional, equipos de transmisión, hardware de STB y software de STB. Para el desarrollo de aplicaciones, estas redes utilizan APIs propietarias, como pueden ser MediaHighway, Open TV o d-box Network.

No obstante, con el objetivo de dar una integración adecuada a estos nuevos servicios en el mercado de la TV, es necesario estandarizar las tecnologías utilizadas a lo largo de la cadena ya que sólo los mercados horizontales y las garantías de compatibilidad permiten reducir costes y llegar a tantos usuarios como sea posible. De este aspecto también se ha ocupado recientemente el consorcio DVB.

Una vez definidos los estándares de difusión, el ámbito del proyecto DVB se extendió para trabajar en una API genérica común que permita que se puedan descargar a través de las redes de difusión aplicaciones interactivas y se puedan ejecutar en receptores con implementaciones hardware y software específicas de cualquier fabricante. Este estándar reciente se conoce como MHP (Multimedia Home Platform) [1], define un contexto para ejecutar aplicaciones y una interfaz software (la API MHP) para que estas aplicaciones puedan acceder a los recursos hardware de cualquier tipo de receptor, desde STBs hasta PCs multimedia. El núcleo de la especificación MHP está basada en una plataforma denominada DVB-J, que incluye una máquina virtual Java (JVM), tal como la especifica Sun Microsystems. Una serie de paquetes java (también llamados APIs) proporcionan interfaces entre la JVM y las características y funciones de un receptor DVB y la red a que esté conectado. También se especifican el formato de los contenidos, los protocolos de red y la señalización.

La arquitectura de un receptor MHP se basa en un sistema operativo y diversos elementos (middleware) situados sobre él y por debajo de las aplicaciones MHP. En este artículo, presentamos nuestra experiencia diseñando e implementando un prototipo de receptor MHP utilizando dos posibilidades en lo que a sistema operativo se refiere: una basada en RT-Linux y la otra en un sistema operativo de desarrollo propio.

El artículo se organiza de la siguiente forma: la sección 2 proporciona una visión general de la arquitectura, tanto hardware como software del receptor digital. La sección 3 presenta las características que debe presentar el sistema operativo del receptor y las dos alternativas consideradas. En la sección 4 se comparan estas dos alternativas en base a una serie de características que consideramos de interés y también se presenta una experiencia consistente en implementar un algoritmo de planificación de tareas sobre ambos sistemas. Por último, en la sección 5 se presentan las principales conclusiones obtenidas.

2 Receptor digital de televisión MHP (Set-Top Box)

La tecnología para la producción y transmisión de televisión de alta definición está disponible en la actualidad, si bien uno de los principales problemas que encuentra su expansión es que la práctica totalidad de receptores de TV que hay en las casas de los usuarios potenciales son analógicos, lo que exige disponer de un convertidor de señales digitales a analógicas. Como ya se ha indicado, a este convertidor se le denomina Set-Top Box (STB).

Además de recibir y procesar el flujo de transporte y los paquetes de información del sistema, el STB debe proporcionar las facilidades necesarias para proporcionar interactividad, a través del canal de retorno, para las distintas redes a las que acceda. Esta interfaz de acceso permitirá a los operadores de red o a los desarrolladores de software suministrar nuevas aplicaciones y personalizar las propias interfaces de usuario del STB. Un abonado seleccionará el flujo de información indicando sus preferencias a través de un sistema software gráfico de navegación.

La rápida evolución de la tecnología de los microprocesadores y la disponibilidad de un cada vez mayor ancho de banda para la transmisión de señal, bien mediante cable o bien mediante la compresión de señales digitales, posibilita encomendar al STB un papel más importante que el de un simple conversor de señales. Existen tres tipos de STB:

1. Analógico: Proporciona recepción de señal de audio y/o vídeo digital.
2. Analógico Inteligente: Además de la recepción de señal de audio y/o vídeo digital, permite también pago por visión, guías electrónicas de programación y algún sistema simple de mensajería.
3. Digital: Permite la recepción y procesamiento de señales digitales.

El STB permite además transformar la televisión en un servicio interactivo. Para ello es necesario un canal de retorno que permita una comunicación bidireccional, para el que existen tres posibilidades: la red telefónica conmutada pública a través de un módem, un canal inalámbrico que permite la transmisión de datos al centro emisor y a través de la red de cable.

Dependiendo del coste, un STB podrá ofrecer un rango de funciones entre mínimas y avanzadas. Además de la capacidad de proceso del flujo de transporte constituido por los paquetes MPEG, y de la posibilidad de utilizar una interfaz gráfica de usuario y de interacción con las aplicaciones que sea configurable y extensible; además es un requisito básico que el STB disponga de mecanismos de seguridad y autentificación de los abonados, tanto para evitar accesos no autorizados a los servicios como para permitir el desarrollo de sistemas de acceso condicional. El control de acceso será, por tanto, parte integral del sistema software de gestión del dispositivo. La identificación de cada STB requiere un esquema de direccionamiento adecuado. El abonado sólo precisará de un sencillo dispositivo de entrada, como un control infrarrojo, para guiar las operaciones del STB, pero se prevén módulos de control de varios dispositivos periféricos adicionales con los que aumentar las capacidades del equipo, por ejemplo, teclado,

controladores de impresora, un CD-ROM o discos magnéticos de almacenamiento de datos. En la figura 1 se puede observar la arquitectura modular de nuestro prototipo.

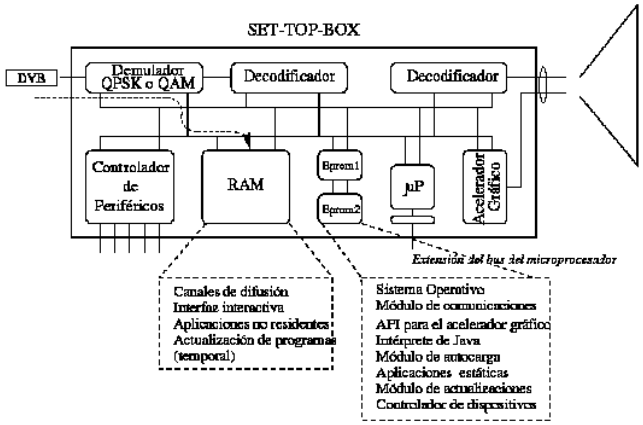


Figura1. Arquitectura general del receptor digital

La arquitectura software de un receptor MHP (figura 2) está formada por un sistema operativo que da soporte a los elementos *middleware* que están situados bajo la API MHP: el gestor de aplicaciones, la torre de protocolos de comunicaciones y la JVM. Además, entre la JVM y la API MHP, se debe incluir un conjunto de APIs (Sun Java, HAVi, DAVIC y DVB). Por otro lado, se debe proporcionar una interfaz gráfica para permitir al usuario interactuar con el sistema. La aplicación que implementa esta interfaz se suele denominar *Home Navigator*.

3 El sistema operativo

Una de las primeras decisiones que debemos tomar a la hora de diseñar a nivel software un STB es la elección del sistema operativo que proporcione soporte al receptor MHP.

A la hora de definir las características de este sistema operativo debemos tener en cuenta las características propias del dispositivo que queremos controlar (STB). A diferencia de los ordenadores de propósito general (ya sean servidores, personales o estaciones de trabajo), los ordenadores (entendiendo por ordenador el conjunto formado por procesador, memoria y dispositivos de entrada salida) utilizados para controlar dispositivos específicos no tienen razón de existencia por sí mismos. Sólo tienen sentido como componentes del sistema del que forman parte. Por este motivo son conocidos como ordenadores embebidos o sistemas embebidos.

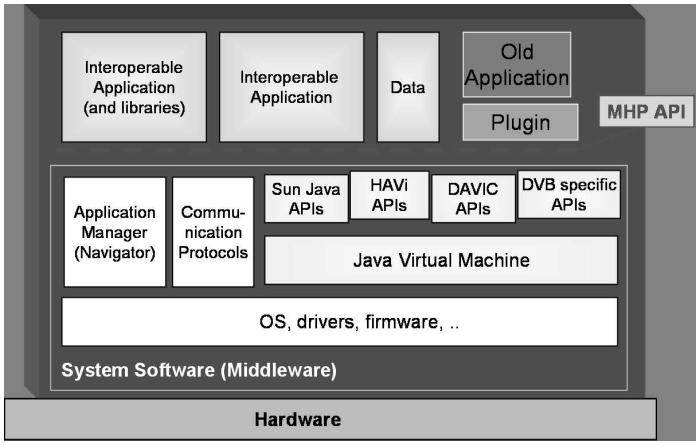


Figura2. Arquitectura software de un receptor MHP

Un sistema embebido debe detectar eventos externos y cambios en su entorno a través de sensores u otros dispositivos de entrada, y realizar las acciones de respuesta adecuadas. Normalmente, también tienen que realizar acciones sobre su entorno de manera cíclica, es decir, determinadas tareas ejecutándose en un sistema embebido deben monitorizar constantemente su entorno y controlar dicho entorno. Es decir, debe presentar características de tiempo real.

Por último, debe tenerse en cuenta que un sistema embebido dispone de unos recursos mucho más limitados que un ordenador de propósito general (menor cantidad de memoria, procesador menos potente, ausencia en general de dispositivos de almacenamiento, etc.). Por tanto, el sistema operativo debe ser lo más simple posible, implementando solamente aquellas funciones que sean estrictamente necesarias para proporcionar las funcionalidades requeridas por las aplicaciones (por ejemplo, no tiene sentido que el sistema operativo proporcione soporte para memoria virtual si el sistema no dispone de dispositivos de almacenamiento). Con esto conseguiremos que el núcleo del sistema sea lo más pequeño posible (normalmente el núcleo de un sistema operativo embebido tiene un tamaño de unos pocos Kbytes. frente a los tamaños en torno a 1 Mbyte o más de los sistemas operativos de propósito general), y además imponga sobrecargas mínimas al sistema, permitiendo disponer de mayor porcentaje de utilización de la CPU para los programas de aplicación.

Por todo lo expuesto, los sistemas operativos utilizados en sistemas embebidos son bastante diferentes de los sistemas operativos utilizados en sistemas de propósito general. Los sistemas operativos para sistemas embebidos deben ser **pequeños sistemas operativos multitarea de tiempo real que impongan pocas sobrecargas al sistema**.

Una vez definidas las características básicas del sistema operativo que necesitamos, podemos plantearnos las siguientes posibilidades:

1. Utilizar un sistema operativo comercial que cumpla las características anteriores.
2. Utilizar un sistema operativo de libre distribución que cumpla las características anteriores.
3. Desarrollar un sistema operativo propio.

Si optamos por las opciones 1 ó 2, la decisión de utilizar como elemento central del STB el circuito TMS320AV7000 de Texas Instruments (TI) restringe la elección del sistema operativo, ya que dicho circuito utiliza un microprocesador ARM7TDMI, y por lo tanto sólo podremos elegir entre sistemas operativos de los que existan versiones disponibles para dicho procesador. Además se debe tener en cuenta que aunque el sistema final se implementará sobre este procesador, el desarrollo del prototipo se realizará sobre una plataforma tipo PC.

Dentro de la primera opción cabe señalar que el Sistema de Desarrollo proporcionado por Texas Instruments para el circuito AV7000 incluye el sistema operativo de tiempo real pSOSystem de Integrated Systems Inc. (ISI), el cual incluye soporte no sólo para el microprocesador utilizado, sino también controladores para el resto de los elementos que componen dicho circuito (decodificadores, acelerador de gráficos, etc.), por lo que parece la opción más adecuada si se decide utilizar un sistema operativo comercial. No obstante, existen otros sistemas operativos de tiempo real para sistemas embebidos para la familia de microprocesadores ARM7. Todos estos sistemas operativos tienen en común la característica de no estar libres de *royalties* (es decir, por cada STB basado en ese sistema operativo que se venda habrá que pagar una cantidad al fabricante del sistema operativo). Esta característica nos hace desestimar esta opción, ya que uno de los objetivos principales es obtener un producto lo más barato posible para que sea competitivo en el mercado.

Otra desventaja importante de este tipo de sistemas es la no disponibilidad del código fuente del sistema operativo.

Dentro de la segunda opción, el único sistema operativo de tiempo real de libre distribución que existe para la familia de procesadores ARM es Linux con extensiones para tiempo real.

La tercera opción consiste en desarrollar un sistema operativo de tiempo real propio y adaptado a las necesidades del STB. Esto supondrá unos costes de desarrollo iniciales más elevados, pero permitirá obtener un producto final de menor coste respecto a la utilización de un sistema operativo comercial.

Para nuestro prototipo hemos considerado las dos últimas posibilidades. En las siguientes secciones presentamos de manera más detallada ambas alternativas.

3.1 RT-Linux

Hoy en día, Linux puede ser una posible opción a considerar para un sistema embebido. Aunque Linux no fue diseñado originalmente como un sistema operativo de tiempo real, sino como un sistema operativo de propósito general para

servidores y ordenadores personales, su popularidad y potencialidad está llevando a muchos desarrolladores a empezar a utilizarlo en cada vez más sistemas, incluyendo sistemas embebidos y de tiempo real. Gracias a la disponibilidad del código fuente, Linux está respondiendo rápidamente a las demandas del mercado, tanto en términos de tamaño reducido para sistemas embebidos como en prestaciones de tiempo real.

Es en este contexto donde aparecen algunas soluciones como RT-Linux [2], que proporciona cumplimiento de tiempos de respuesta de caso peor determinista mediante un pequeño núcleo de tiempo real, ejecutando Linux convencional sobre este núcleo utilizando el tiempo libre de procesador. Con esta solución, el sistema puede responder a eventos en tiempo real al mismo tiempo que proporciona todos los servicios de Linux a las aplicaciones.

Estos cambios en Linux hacen posible que se esté utilizando como sistema operativo en un número creciente de dispositivos diferentes. En la actualidad, Linux se está convirtiendo en una solución cada vez más atractiva para dispositivos como Set-Top Boxes, equipos multimedia o dispositivos móviles. Muchos fabricantes de equipos electrónicos de consumo ya experimentan con él (dentro del campo de la ITV, podemos destacar la *TV Linux Alliance*, creada en junio de 2001 por 24 fabricantes para definir un entorno estándar basado en Linux para el mercado de STB digitales [3]. Se espera que esta especificación esté disponible a finales de 2002).

3.2 Sistema Operativo propio

Si optamos por el desarrollo de un sistema operativo propio para el STB, debemos decidir cómo abordarlo. Hemos decidido utilizar la especificación ITRON (que es parte del proyecto TRON). ITRON es una norma que se empezó a definir en el año 1984 en Japón para el desarrollo de sistemas operativos de tiempo real para sistemas embebidos, dentro de un consorcio formado por un conjunto de empresas y universidades, estando actualmente publicada la versión 3.0 de la norma [4].

Las especificaciones de ITRON son de libre uso, y entre el 30% y 40% de los equipos de consumo fabricados en Japón las utilizan como base para sus sistemas operativos embebidos. También la utilizan fabricantes de Estados Unidos y Europa. Podemos afirmar, por tanto, que es un estándar de facto, existiendo más de 45 implementaciones registradas para más de 40 procesadores diferentes (desde 8 bits a 64 bits).

El sistema operativo que hemos desarrollado se denomina *Eros* (*Embedded Real-time Operating System*) e implementa las funciones necesarias para nuestro prototipo de las definidas en la especificación ITRON. En concreto se han implementado las funciones que aparecen subrayadas en la tabla 1. El núcleo de la versión actual del sistema operativo tiene un tamaño aproximado de 8 Kb.

Tabla1. Funciones soportadas por la especificación ITRON

1. Gestión de tareas:
   - Gestión directa de tareas y de su estado.
2. Sincronización interna en las tareas:
   - Función de sincronización de una tarea dentro de la propia tarea.
3. Sincronización y comunicación entre tareas:
   - Se proporcionan tres funciones de sincronización y comunicación entre tareas independientes de las tareas: semáforos, señalizadores de eventos y buzones.
4. Mecanismos adicionales de sincronización y comunicación:
   - Existen dos funciones avanzadas de sincronización y comunicación entre tareas que son: colas de mensajes y rendezvous.
5. Gestión de interrupciones:
   - Funciones para definir rutinas de atención a interrupciones externas.
   - Funciones para permitir e inhibir interrupciones externas.
6. Gestión de memoria:
   - Funciones para la gestión software de memoria dinámica.
7. Gestión de tiempo:
   - Funciones para la gestión del reloj del sistema.
   - Función para retrasar tareas.
   - Funciones para temporizadores.
8. Gestión del sistema:
   - Funciones para modificar o acceder a las variables del sistema.
9. Soporte de red:
   - Funciones de soporte y gestión para una red débilmente acoplada.

4 Comparativa

Para realizar una comparativa de las dos opciones contempladas hemos identificado una serie de características que consideramos de interés y que nos permiten realizar una comparación cualitativa.

También hemos realizado un experimento consistente en adaptar un algoritmo de planificación a ambos sistemas operativos con el fin de realizar una comparación cuantitativa del esfuerzo necesario para adaptar cada uno de los dos sistemas operativos a las necesidades específicas de un desarrollador.

4.1 Comparación cualitativa

En esta sección se presenta una valoración de diferentes características de interés para nuestros objetivos de las dos soluciones evaluadas.

- **Portabilidad.**
  Puesto que vamos a implementar nuestro prototipo sobre diferentes plataformas hardware, un aspecto importante es la portabilidad.
  Obviamente un sistema operativo nunca será totalmente portable, ya que ello significaría una excesiva virtualización del hardware, lo cual supondría la imposibilidad de obtener las máximas prestaciones del procesador y demás

dispositivos sobre los que se implemente, y de tener un buen comportamiento en tiempo real.
Por tanto, en un sistema operativo siempre habrá que realizar cambios para adaptarlo a un procesador o plataforma hardware específicos. La portabilidad en este caso significa que los cambios que haya que hacer sean mínimos y sencillos.
En cuanto al sistema RT-Linux, aunque en principio existen versiones de Linux para muchos de los procesadores del mercado, no ocurre lo mismo con las extensiones de tiempo real, que además están escritas en parte en código máquina para procesadores de Intel. Por tanto, habrá que realizar las modificaciones necesarias sobre el código de las extensiones de tiempo real.
En cuanto al sistema operativo propio, la especificación ITRON distingue claramente entre los aspectos que deben ser normalizados, y que son independientes de la arquitectura hardware, y aquellos que deben ser optimizados en base a la arquitectura específica que se esté utilizando. Entre los aspectos que se normalizan están los algoritmos de planificación de tareas, nombres y funciones de las llamadas al sistema, nombres, orden y significado de los parámetros, y nombre y significado de los códigos de error. Entre los aspectos que no se estandarizan, porque pueden influir en las prestaciones del sistema, podemos indicar, por ejemplo, los mecanismos de atención a interrupciones o el tamaño de los parámetros, que serán diseñados específicamente para cada implementación. Esta distinción permite aislar claramente el código dependiente de la máquina facilitando enormemente la portabilidad.

– **Posibilidad de adaptación a las aplicaciones específicas.**
Cuando un sistema operativo se va a utilizar embebido en un dispositivo para un uso específico, como puede ser nuestro prototipo, es sumamente interesante el poder seleccionar fácilmente qué componentes y facilidades se compilarán en el núcleo (sólo se deberían compilar las necesarias para las aplicaciones a ejecutar) con el fin de tener unas sobrecargas mínimas debidas al sistema operativo, y que éste sea del menor tamaño posible.
En este sentido, aunque Linux tiene una cierta capacidad de configuración del núcleo, existen muchos componentes y funcionalidades que se mantienen en cualquier configuración como parte integral del núcleo y que pueden no ser necesarias para un dispositivo específico (determinados mecanismos de sincronización y comunicación entre procesos, gestión de memoria virtual, etc.) Esta característica viene impuesta por la propia naturaleza de Linux, que nació como un sistema operativo de propósito general para ordenadores. Esto nos lleva a una capacidad limitada de decisión sobre las funcionalidades que debe proporcionar el sistema operativo, y esta falta de adaptabilidad nos lleva a tener núcleos grandes con funciones que muchas veces no son necesarias.
En cuanto a nuestro sistema operativo, la propia especificación ITRON permite cambiar las especificaciones del núcleo y los métodos de implementación interna en base a las funciones del núcleo y las prestaciones que necesiten las aplicaciones, con el objetivo de aumentar el rendimiento global del sistema.
En un sistema embebido, el código objeto del sistema operativo se genera

de manera separada para cada aplicación o conjunto de aplicaciones (sistema), por tanto esta posibilidad de adaptación es especialmente adecuada (ver figura 3).

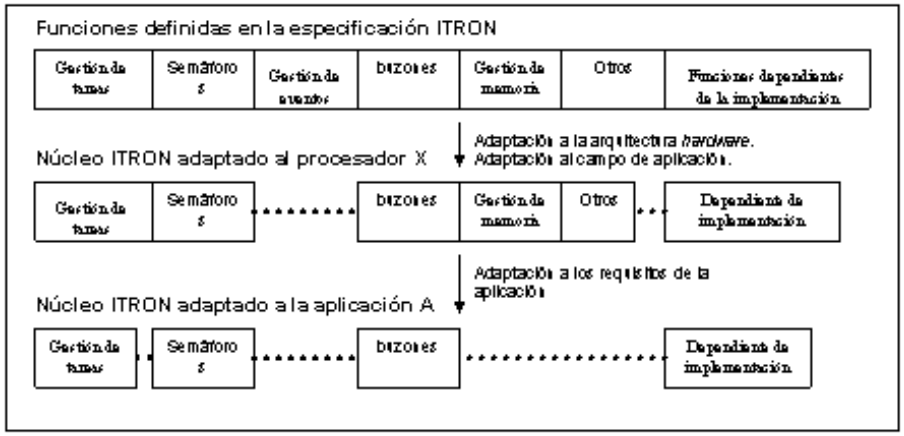


Figura3. Adaptación de la especificación ITRON

En ITRON, esta adaptación se ha realizado haciendo que las funciones proporcionadas por el núcleo sean tan independientes unas de otras como sea posible, permitiendo seleccionar solamente aquellas funciones que son necesarias para cada aplicación. En la práctica, la mayoría de los núcleos basados en la especificación ITRON se proporcionan en forma de librería. La selección de funciones del núcleo se hace simplemente enlazando esa librería con la aplicación, con lo que sólo las funciones necesarias son incorporadas al sistema. Además, cada llamada al sistema proporciona una sola función, haciendo más sencillo incorporar sólo las funciones necesarias.
En este punto se ve claramente la diferencia entre un sistema operativo de propósito general adaptado a su utilización en sistemas embebidos (RT-Linux) y un sistema operativo diseñado desde el principio teniendo en cuenta su carácter embebido (especificación ITRON).

– **Funcionalidad disponible**
Aunque en ITRON las primitivas proporcionadas por el núcleo no están limitadas a una pequeña cantidad, sino que proporcionan un amplio conjunto de funciones (ver tabla 1), en este punto la ventaja es clara para Linux, ya que su grado de desarrollo y madurez, así como su carácter de sistema operativo de propósito general hacen que proporcione una funcionalidad muy superior a la especificada en la norma ITRON.
Sin embargo, está funcionalidad sólo esta disponible para las aplicaciones que no presentan características de tiempo real, ya que los procesos de tiempo real se ejecutan a un nivel inferior a Linux y no pueden utilizar la mayoría

de funciones del núcleo y de librería al no estar diseñadas para ejecutarse en tiempos deterministas. Esta limitación, no obstante, no es muy importante, ya que normalmente las tareas complejas no suelen presentar necesidades de tiempo real estricto. Las tareas de tiempo real suelen ser simples y sin necesidades de funcionalidades complejas.

- **Componentes software (middleware).**

  Una vez normalizado el núcleo del sistema operativo, el Comité Técnico del proyecto ITRON, ante la situación actual en que los sistemas embebidos cada vez son más complejos y de mayor tamaño, decidió crear una serie de estándares de los que denomina componentes software (o middleware), que permitan integrar fácilmente con un núcleo de sistema operativo basado en la especificación ITRON, nuevas funcionalidades que faciliten la creación de esos sistemas complejos, sin perder las características (sobre todo de tiempo real) del propio núcleo.

  Se pretenden normalizar componentes software tanto genéricos como para campos de aplicación específicos (como puede ser, por ejemplo, el campo de sistemas embebidos en automóviles, para el que ya se está normalizando una interfaz para componentes software).

  Dentro de los componentes software genéricos, actualmente están normalizadas dos interfaces dentro del proyecto ITRON: una API de la torre de protocolos TCP/IP, y el entorno de ejecución Java (JTRON). Estas dos especificaciones se utilizan en nuestro prototipo.

  Sin embargo, a pesar de este esfuerzo de normalización de componentes, la ventaja de Linux en este aspecto vuelve a ser muy superior, ya que no es necesario desarrollar estos componentes al estar ya disponibles, con el consiguiente ahorro de tiempo y coste de desarrollo. Además, la existencia de diversas versiones de los mismos componentes (por ejemplo, existen multitud de Java VM) permite elegir la más adecuadas a las necesidades de nuestro prototipo, comparar prestaciones entre implementaciones, etc. y todo ello sin necesidad de acometer ninguna labor de desarrollo.

- **Controladores de dispositivo**

  La especificación ITRON no contempla la gestión de dispositivos de entrada/salida, ya que en sistemas embebidos hay muy pocos dispositivos de este tipo comunes entre sistemas (así por ejemplo, un STB no tendrá ningún dispositivo externo en común con un sistema embebido de gestión de los sistemas eléctricos y electrónicos de un automóvil).

  Aquí de nuevo aparece una clara ventaja de Linux, ya que existen multitud de controladores para prácticamente todos los dispositivos habituales del mercado, pudiéndose utilizar directamente muchos de ellos sin ninguna necesidad de desarrollo. No obstante, se debe tener en cuenta que estos controladores sólo pueden ser empleados por tareas que se ejecuten sobre Linux y, por tanto, no tengan necesidades de tiempo real estricto, lo cual, en este caso sí que puede suponer una limitación.

4.2 Comparación cuantitativa: adaptación de un algoritmo de planificación

Con el fin de valorar experimentalmente las ventajas e inconvenientes de cada una de las dos alternativas consideradas, hemos analizado la capacidad de adaptación a las necesidades del sistema final de cada uno de ellas, modificando el algoritmo de planificación de tareas, que en un principio era *Rate Monotonic* en ambos sistemas, e introduciendo un algoritmo de planificación basado en prioridades dinámicas que afectara tanto a las tareas de tiempo real críticas como no críticas. El algoritmo seleccionado fue EDF-MTR [5], que es un algoritmo basado en el *Earliest Deadline First (EDF)* pero que mejora el tiempo de respuesta de las tareas no críticas. Este algoritmo presenta un comportamiento mucho más próximo al del algoritmo óptimo (que no es implementable en la práctica debido a su gran complejidad) que al de EDF ejecutando tareas no críticas en segundo plano. Además este comportamiento se consigue con un aumento muy pequeño de las cargas administrativas respecto a EDF (aproximadamente un 10%).

Elegimos este algoritmo porque no sólo hay que modificar el mecanismo de planificación de las tareas de tiempo real críticas (lo cual es bastante sencillo en RT-Linux, al ser un módulo) sino que también hay que modificar el mecanismo de planificación del resto de las tareas (es decir, el mecanismo de planificación clásico de Linux).

Las modificaciones en el sistema operativo basado en ITRON fueron muy sencillas, ya que la cantidad de código que forma el núcleo es pequeña y manejable y está perfectamente identificada y aislada la parte de código a modificar.

En cuanto a Linux, la modificación de la planificación de tareas de tiempo real también fue muy simple, al ser la cantidad de código de las extensiones de tiempo real pequeña y estar el código del planificador convenientemente aislado. Sin embargo, la modificación del planificador de Linux (el utilizado por las tareas no críticas) supuso un esfuerzo mucho mayor ya que el código del núcleo de Linux es complejo y bastante disperso en el sentido de que muchos de sus componentes interactúan con otros. Así por ejemplo, el código del planificador actúa sobre las Tablas de Control de Procesos (TCB) que son mucho más complejas que las de las tareas de tiempo real, ya que incluyen información sobre mecanismos de sincronización y comunicación, gestión de memoria virtual, etc. que debe ser adecuadamente gestionada por el planificador y que no existe ni en el núcleo de ITRON ni en la parte de tiempo real de RT-Linux.

Otra dificultad importante que se presenta es la del tratamiento totalmente independiente que se hace en RT-Linux de las tareas de tempo real y de las propias tareas de Linux (tablas de control de procesos independientes, etc.), lo que dificulta sobremanera una adecuada gestión conjunta de ambos tipos de tareas, tal como se debe hacer para implementar el algoritmo elegido. Este problema se debe a que las extensiones de tiempo real se han diseñado como una nueva capa que se instala entre el hardware y el propio Linux, en vez de integrarse en el propio núcleo de Linux.

Cuantitativamente hablando, los resultados obtenidos son los siguientes:

	RT-Linux	**Eros**
Tiempo de implementación	8 horas	40 horas
Número de ficheros modificados	2	17
Tamaño del núcleo resultante	8 Kb.	329 Kb.

5 Conclusiones

Tras nuestra experiencia con las dos alternativas propuestas como sistema operativo de nuestro prototipo de receptor, no podemos concluir que ninguna de ellas sea claramente superior a la otra, aunque sí que ambas son alternativas válidas.

RT-Linux presenta como principal ventaja su robustez y estabilidad, la rápida disponibilidad de soporte de nuevos dispositivos hardware debido a la gran cantidad de desarrolladores existentes y, para nuestros intereses, sobre todo la disponibilidad de una gran cantidad de middleware (JVM, protocolos de comunicaciones, etc.) lo que nos permite que el diseño e implementación del receptor MHP se limite a desarrollar el API MHP sobre una JVM existente.

Sin embargo, presenta una serie de inconvenientes, que precisamente son los puntos fuertes de un desarrollo propio. Esto es debido a que las características de tiempo real y sistema embebido de RT-Linux se introducen como un "parche" sobre un sistema de propósito general y por tanto no se han tenido en cuenta en el diseño del propio sistema operativo, cosa que sí ocurre con el sistema operativo desarrollado.

Estas ventajas del sistema operativo basado en la especificación ITRON frente a RT-Linux son, principalmente: diseño orientado desde el principio al tipo de sistemas que estamos considerando; portabilidad y adaptabilidad sencilla al hardware específico; y sencillez, lo que se traduce en un núcleo pequeño y sólo con la funcionalidad necesaria. El principal problema del sistema operativo basado en ITRON es la necesidad de diseñar e implementar todo el middleware y los controladores de dispositivo necesarios para el sistema.

Podríamos concluir diciendo que consideramos RT-Linux más adecuado para el desarrollo de un prototipo de manera rápida y sobre una plataforma de propósito general, con componentes de base suficientemente probados y que nos permiten centrarnos en el desarrollo de las capas superiores de la arquitectura software del sistema y el sistema operativo de desarrollo propio más adecuado para el sistema final que se ejecutará sobre un hardware específico.

Referencias

1. DVB Consortium, "Multimedia home platform (MHP) 1.1," July 2001.
2. Michael Barabanov, "A linux-based real-time operating system," M.S. thesis, New Mexico Institute of Mining and Technology, Socorro, New Mexico, USA, June 1997.
3. "http://www.tvlinuxalliance.org," .
4. "http://http://www.itron.gr.jp," .
5. Manuel Ramos Cabrer, *Una arquitectura para sistemas inteligentes adaptativos basada en el modelo de pizarra*, Ph.D. thesis, Universidad de Vigo, June 2000.

ADA_MAST: Herramienta para el Desarrollo de Aplicaciones Ada Distribuidas de Tiempo Real [1]

Julio L. Medina, José M. Drake, J. Javier Gutiérrez, Michael González Harbour

Avda. de los Castros s/n, 39005 Santander - España
{medinajl, drakej, gutierjj, mgh}@unican.es

Resumen. Se describe una metodología para el modelado de tiempo real de aplicaciones Ada 95 construidas de acuerdo con los anexos de tiempo real (D) y distribuido (E) del estándar. El modelo se formula utilizando UML y puede ser soportado por herramientas CASE estándar orientadas a objetos. El modelo de tiempo real que resulta es apropiado para ser procesado por el conjunto abierto de herramientas que ofrece el entorno MAST. Este modelo representa independientemente la plataforma que ejecuta la aplicación, los componentes lógicos software con que se construye y las situaciones o escenarios de tiempo real con que puede operar y que son el objeto del análisis. A través de un ejemplo de sistema distribuido, se muestra como utilizar la metodología y las herramientas en fases de diseño de la aplicación (por ejemplo para la asignación óptima de prioridades) y en fases de análisis (por ejemplo, estudio de su planificabilidad, cálculo de holguras, etc.). El modelo de la invocación de componentes distribuidos, es gestionado de forma automática por la herramienta, y es establecido de forma diferente si la invocación es local o remota..

1. Introducción

Este trabajo propone una metodología para describir con UML sobre una herramienta CASE, el modelo de tiempo real de aplicaciones Ada distribuidas. Los elementos de modelado que se definen, permiten modelar aplicaciones distribuidas sobre una implementación de Ada 95 que soporte los anexos D de sistemas de tiempo real y E de sistemas distribuidos del estándar [1] y proporcionan un modelo que puede ser procesado por el conjunto de herramientas que se ofrecen en el entorno MAST (Modeling and Analysis Suite for Real-Time Applications) [8][9]. La reusabilidad de módulos software a través de la programación orientada a objetos es una de las ventajas que supone el uso de Ada en aplicaciones de tiempo real. Sin embargo, para diseñar módulos software de tiempo real reusables, se necesita tener capacidad para modelar su comportamiento de tiempo real, al igual que se describe su funcionalidad. Cuando se modelan aplicaciones que además son distribuidas, se tiene que incluir en el modelo el efecto de las comunicaciones y planificar la transferencia de los

[1] Este trabajo está financiado por la Comisión Interministerial de Ciencia y Tecnología mediante los proyectos TIC99-1043-C03-03 y 1FD 1997-1799 (TAP)

mensajes a través de la red de comunicaciones al igual que se planifican las actividades que se ejecutan en los procesadores.

Aspectos relevantes de la propuesta que se presenta son:

- Tal como se sugiere en [9], plantea modelos independientes para la plataforma (procesadores, redes de comunicación, recursos del sistema operativo, dispositivos externos, etc.), los componentes lógicos de la aplicación (requisitos de procesamiento, recursos comunes, otros componentes internos), y las situaciones de tiempo real que definen la funcionalidad esperada del sistema en sus diversos modos de operación (secuencias de eventos, transacciones de tiempo real, carga de trabajo, requisitos temporales).
- Modela el comportamiento de tiempo real de las entidades lógicas de Ada (temporización, concurrencia, sincronización, etc.), de manera que el modelo de tiempo real sigue la misma estructura del código Ada original.
- Facilita la generación del modelo de los diversos componentes lógicos de más alto nivel mediante la instanciación de modelos genéricos parametrizados. Cuando se combinan los modelos de todos los componentes que intervienen en cada situación de tiempo real, y se asigna valores concretos a los parámetros del modelo, se obtiene el modelo analizable de todo el sistema para esa situación de tiempo real.
- El modelo de comunicaciones que representa la codificación, transmisión, entrega y decodificación de argumentos en las llamadas a procedimientos remotos, se incorpora de manera automática cuando un procedimiento que se declara como remoto se invoca desde un componente que se encuentra asignado a un nodo diferente.
- Al igual que la descripción funcional y estructural de los componentes del software de la aplicación, el modelo de tiempo real se describe en UML, y se hace dentro de una nueva vista de la aplicación. Los componentes del modelo, así como su interconectividad se han descrito formalmente mediante un metamodelo.

En el siguiente apartado se apuntan algunos antecedentes de la metodología que proponemos y las herramientas sobre las que se apoya. En el apartado 3 se muestra a través de secciones del metamodelo, los tipos de componentes de modelado básicos, su semántica y posibilidades de asociación. En el apartado 4 se plantean algunos aspectos sobre la correspondencia de módulo construidos con sentencias Ada y su modelado. Un ejemplo de diseño y validación de una aplicación a través de la metodología se presenta en el apartado 5. Finalmente resumimos algunas conclusiones y líneas de trabajo futuro.

2 Antecedentes, herramientas y técnicas de soporte

Tanto desde el punto de vista semántico como conceptual, el metamodelo que soporta la descripción en UML de la metodología que se propone, es una especialización del propuesto con la herramienta UML-MAST [9] que fue desarrollado para modelar aplicaciones diseñadas con tecnologías orientadas a objetos. Y ésta a su vez es una especialización de la herramienta MAST [8], propuesta para el modelado de sistemas de tiempo real más generales, gobernados por eventos. El conjunto de herramientas disponibles es compartido por todas ellas y actualmente se dispone de herramientas

para realizar análisis de planificabilidad (tanto por el método basado en offsets[11][12] como por el método clásico), realizar asignación óptima de prioridades (por los métodos de templado simulado y Linear HOPA) o calcular holguras. Estas herramientas son además aplicables tanto a sistemas mono y multiprocesador, como a sistemas distribuidos, y permite emplear estrategias de planificación basadas en prioridades fijas, expulsoras y no expulsoras, planificación de rutinas de interrupción, servidores esporádicos y servidores de atención periódica.

Desde su inicio, MAST fue concebido como un entorno abierto que proporciona las especificaciones y los recursos necesarios para que los investigadores que trabajan en tiempo real puedan integrar y comparar las nuevas herramientas que desarrollen.

Por otra parte, el concepto de **Componente,** que es central en el modelo, corresponde al que se propone en [13] como elemento principal para el modelado de componentes de tiempo real. En la metodología que se presenta se le ha especializado para representar los tipos básicos de estructuras contenedoras de componentes Ada.

En cuanto al tipo de aplicaciones que se pretende modelar, debemos destacar que los anexos D y E del estándar de Ada 95 han sido pensados y descritos de manera independiente, por lo cual el desarrollo de aplicaciones distribuidas que a la vez deban satisfacer requisitos de tiempo real no está formalmente soportado en Ada 95 [4]. Nuestro grupo ha propuesto un esquema de priorización para las llamadas a procedimientos remotos en [6] y ha apuntado ciertas características de interés para facilitar el desarrollo de sistemas distribuidos de tiempo real en Ada en [7], que lo harían potencialmente más eficiente y fácil de utilizar que otros estándares como CORBA de tiempo real. Por tanto, para que los mecanismos de modelado y análisis que se proponen aquí sean aplicables, presuponemos que la plataforma sobre la que se implementa la aplicación a modelar satisfaga las propuestas de [6] y [7]. Actualmente los compiladores Ada que implementan el anexo D para sistemas monoprocesadores son bastante utilizados, sin embargo, el anexo E de sistemas distribuidos va siendo implementado de manera más paulatina; una de estas implementaciones es GLADE [3], la cual es actualmente parte del compilador GNAT, desarrollado por Ada Core Technologies (ACT) [5] y que es además de distribución libre.

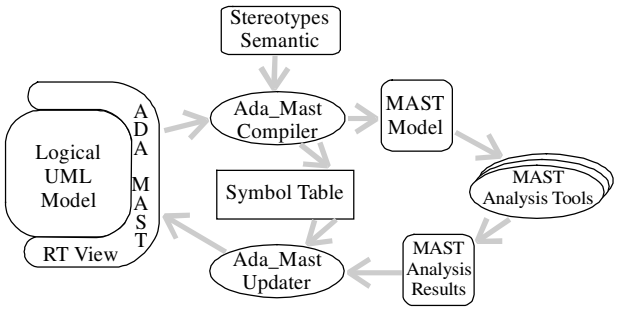


Fig. 1. Forma en que se implementa el paradigma del procesado de modelos con ADA_MAST

En ADA_MAST se sigue un paradigma de procesado de modelos similar al que se describe en [14]. La figura 1 ilustra este proceso. Para procesar el modelo, se compila el modelo UML de tiempo real a un modelo mas general basado en el entorno MAST (MAST Model). La compilación es necesaria ya que con la especialización de la

metodología se incrementa la carga de información semántica, y la generación del modelo de bajo nivel no sólo requiere una codificación de la información contenida en el modelo, sino una incorporación de los patrones introducidos a través de la semántica de los componentes que se utilizan. La compilación genera también una tabla de símbolos que será utilizada para hacer corresponder los resultados que se obtengan de la aplicación de las herramientas con los entes del modelo original a que corresponden. Estos resultados se incorporan al modelo UML en la vista de tiempo real, de manera que el modelo va evolucionando hasta representar la situación de tiempo real del sistema tal como el diseñador la requiere. Los elementos de la vista de tiempo real se designan por medio de estereotipos asignados a elementos estándar UML.

3 Metamodelo de la vista de tiempo real

Siguiendo la dinámica de representación de UML_MAST, el modelo de tiempo real de una aplicación basada en componentes se puede presentar en tres secciones, que modelan separadamente la plataforma, los componentes lógicos y las situaciones de tiempo real bajo análisis. En [13] se describe el metamodelo de cada sección con cierto nivel de detalle, en este apartado presentamos una especialización de este metamodelo que permite soportar de manera sencilla el modelado de aplicaciones Ada sobre la base de los conceptos ofrecidos por MAST.

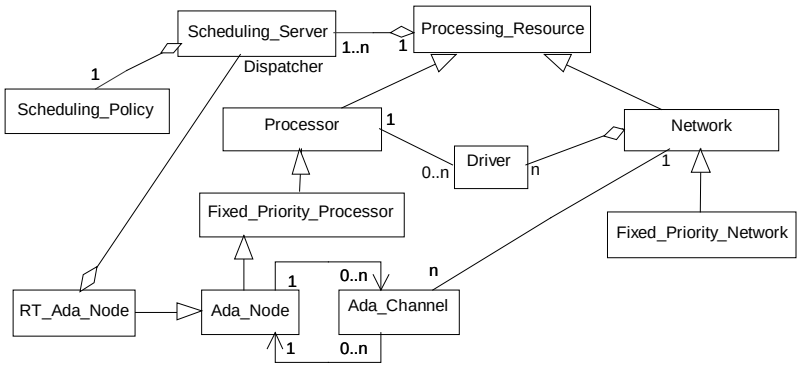


Fig. 2. Sección del metamodelo que describe el modelo de la plataforma

El modelo de la plataforma: Modela la capacidad de procesamiento y las restricciones operativas de los recursos de procesamiento hardware y software que constituyen la plataforma sobre la que se ejecuta el sistema. Estos son recursos tales como procesadores (lo que incluye: capacidad de procesamiento, planificador, threads, temporizadores) y redes de comunicación (capacidad y modo de transmisión, planificación de mensajes, sobrecarga debida a los drivers), y la forma en que están interconectados (qué procesadores comunican a través de qué redes de interconexión).

En la figura 2 se muestra un extracto del metamodelo de la plataforma tal como se define para la metodología que se presenta.

La clase Ada_Node, representa un tipo de procesador basado en prioridades fijas, que puede disponer de canales de comunicación que le enlacen a otros procesadores a fin de invocar procedimientos de acceso remoto en ellos. La clase RT_Ada_Node es equivalente a la anterior, pero además puede exportar estos servicios al disponer de un thread destinado a invocar localmente estos procedimientos una vez demandados desde el exterior.

Modelo de los componentes lógicos: describe el comportamiento de tiempo real de los componentes Ada empleados en la aplicación. Por componente software se designa aquí a cualquiera de los módulos mediante los cuales se distribuyen las operaciones lógicas de la aplicación. Se emplea para modelar paquetes, bien sean pasivos (de librería) o clases (tagged types), tareas, el programa principal (main), etc.

El modelo de un componente software describe: la cantidad de procesamiento que requiere la ejecución de las operaciones de su interfaz, los componentes lógicos de los que requiere servicios, los procesos o threads que crea para ejecutar sus operaciones, los mecanismos de sincronización que requieren sus operaciones, los parámetros de planificación definidos explícitamente en el código y los estados internos relevantes a efecto de especificar y validar requisitos de tiempo real. En la figura 3 se muestra la estructura jerárquica de componentes Ada propuesta.

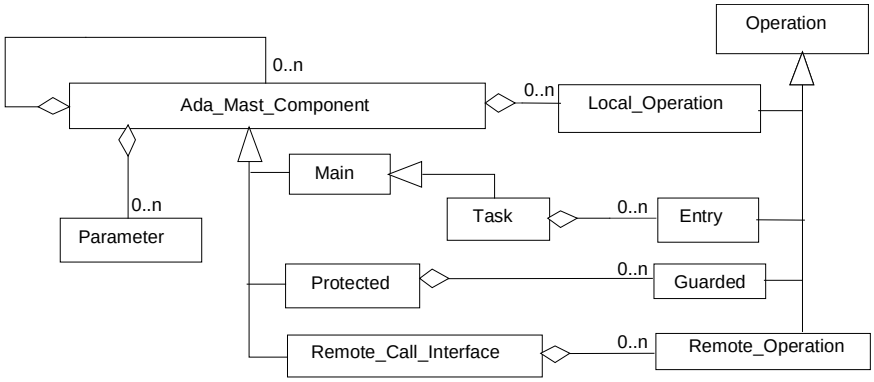


Fig. 3. Sección del metamodelo de componentes lógicos de ADA_MAST.

Ada_Mast_Component es la clase principal del modelo de componentes lógicos de ADA_MAST y agrupa los modelos de comportamiento temporal de todas las operaciones definidas en la interfaz del componente (especificación Ada); para las operaciones simples, emplea un conjunto de atributos que indican la cantidad de procesamiento que requieren del procesador en que se ejecutan, mientras que si son compuestas, su modelo representa la secuencia de operaciones que tiene incluidas. Puede actuar también como contenedor para otros componentes agregados en él, los cuales son instanciados (es decir incorporados al modelo de la situación de tiempo real en que aparecen a partir de su especificación) junto con la instanciación del componente contenedor y se declaran como atributos con el estereotipo <<obj>>.

Puede también tener parámetros, que representan entes que son parte de su descripción pero que están aún por designar, tales como otros componentes referenciados, operaciones, eventos externos, requisitos temporales, etc. Los parámetros se declaran como atributos con el estereotipo <<ref>>. Al momento de instanciar un componente se deben proporcionar los valores concretos de los parámetros que éste tenga definidos. Los parámetros y los componentes agregados de un componente son visibles tanto en su interfaz como en la descripción de sus operaciones y componentes agregados.

Las clases derivadas de Ada_Mast_Component difieren particularmente en cuanto al tipo de operaciones que pueden declarar y las prerrogativas de sus operaciones y parámetros. La clase Main modela el procedimiento principal de una partición Ada y se caracteriza por tener un Scheduling_Server implícito que modela el thread principal. La clase Task representa una tarea Ada y se caracteriza por contener operaciones del tipo Entry, que se emplean para sincronizar el thread de quien invoca la operación con el thread propio del Task. La clase Protected se emplea para modelar los objetos protegidos de Ada, que se caracterizan porque toda las operaciones que declara, bien sean Local_Operations o del tipo Guarded, serán ejecutadas en régimen de exclusión mutua.

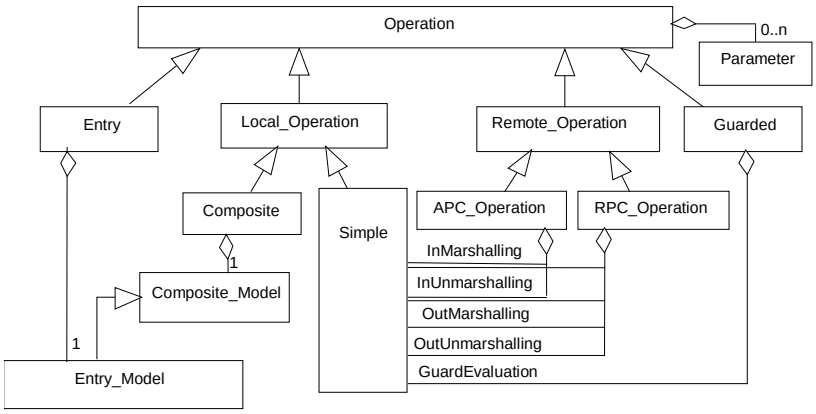


Fig. 4. Jerarquía de operaciones que modelan la interfaz de componentes

Cada operación del tipo Guarded, como son las *entrie*s de un objeto protegido, tiene agregada una operación del tipo Simple_Operation que modela el efecto temporal de la evaluación de la condición de guarda. La clase Remote_Call_Interface modela un componente cuyas operaciones pueden ser invocadas tanto desde la partición local como desde una partición remota. Cada Remote_Operation tiene los atributos necesarios para proporcionar los parámetros de la transmisión e incorpora las operaciones simples que modelan la codificación (*marshalling*) y decodificación (*unmarshalling*) de los argumentos de sus operaciones exportadas. La clase Operation y sus especializaciones se emplean para modelar el comportamiento temporal y los mecanismos de sincronización de los procedimientos y funciones declaradas en la interfaz de los componentes lógicos. La figura 4 muestra esta jerarquía de operaciones.

Modelo de las situaciones de tiempo real: una Real_Time_Situation representa un cierto modo o configuración de operación hardware/software del sistema y la carga de trabajo que soporta para la que se tienen definidos requisitos de tiempo real. Define así el modelo del sistema sobre el cual se han de emplear las herramientas de análisis que sean necesarias. El paso de una determinada situación de tiempo real a otra, es decir, el cambio de modo no está incluido en el modelo, y por tanto cada una se analiza de manera independiente.

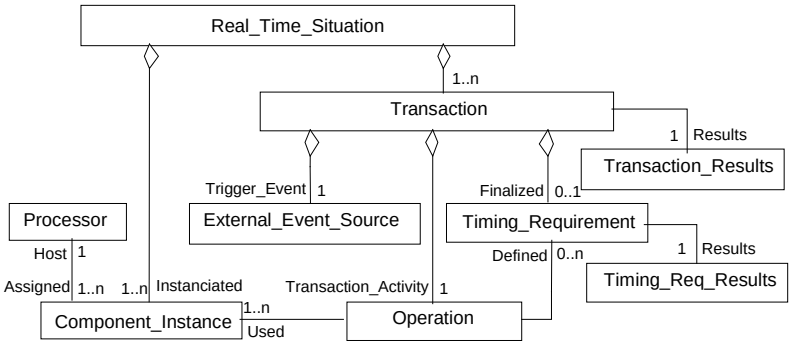


Fig. 5. Sección del metamodelo del modelo de situaciones de tiempo real de ADA_MAST

Una situación de tiempo real se describe por una parte mediante la declaración de todas las instancias de los componentes software que participan en las actividades propias de la configuración que se modela y su correspondiente despliegue sobre la plataforma; por otra, mediante un conjunto de transacciones que modelan la carga del sistema y describen las secuencias no iterativas de acciones que se ejecutan ante la llegada de los eventos externos (External_Event_Source) definidos. Estos eventos además dan la pauta para evaluar si el sistema satisface los requisitos temporales impuestos (Timing_Requirement). La actividad que se inicia ante el disparo del Trigger_Event de una transacción se especifica mediante el enlace Transaction_Activity, y la operación enlazada debe ser exportada por alguna de las instancias declaradas. Una transacción puede tener muchos requisitos temporales, pero hay al menos uno por defecto etiquetado como Finalized, que debe proporcionarse en su declaración y que corresponde a la terminación del grafo de actividad de la transacción. Cualquier otro que se considere relevante y se incluya en algún punto de su descripción, deberá ser exportado por los componentes en que aparece y finalmente ser proporcionado como argumento en la invocación de la Transaction_Activity. Se tienen diversos tipos de eventos de disparo y de requisitos temporales que dan gran versatilidad a los modelos alcanzables, y pueden ser consultados en las referencias sobre MAST [8][9].

Las instancias de clases derivadas de Real_Time_Situation, Transaction, Timing_Requirement o Scheduling_Server tienen todas una instancia asociada cuya clase es una especialización de la clase Results, que contendrá el conjunto de variables en las que recoger los resultados que las herramientas de análisis de planificabilidad proporcionan.

4 Modelo de tiempo real de los componentes Ada

Aunque la metodología de modelado y análisis que se presenta es en principio independiente del lenguaje que realmente se emplee para programar, y puede aplicarse al modelado de un amplio espectro de sistema de tiempo real, la semántica de los componentes de modelado que se han definido, la sintaxis y las convenciones seguidas están específicamente pensadas para programas diseñados y codificados con Ada. En estos casos, su aplicación es más sencilla y automatizable.

El modelo se adapta a la estructura de las aplicaciones Ada: Las instancias de Ada_Mast_Component permiten modelar el comportamiento temporal de *packages* o *tagged classes* que son los elementos estructurales básicos de una arquitectura Ada y en él se describen y se declaran respectivamente los modelos de tiempo real de los procedimientos y funciones públicos del package o clase Ada que modela, y los componentes (otros *packages*, *protected objects*, *tasks*, etc.) que están declarados en el *package* o clase Ada que éste modela, ya sea en la parte pública, en la privada o en el cuerpo, y que son relevantes para modelar su comportamiento temporal.

Un Ada_Mast_Component modela tan sólo el código incluido en la estructura lógica que describe. Así, si un *package* presenta dependencia de otros, no incluye en su modelo el de los *packages* de que depende, únicamente hace referencia a ellos.

Modela la concurrencia que introducen las tareas Ada: Los componentes con estereotipo Task modelan el thread que se introduce con cada tarea Ada. Por cada instancia de un Task se introduce de forma implícita un Scheduling_Server que se asocia al procesador en que se instancia el componente. La sincronización entre actividades de diferentes threads sólo se localiza en los modelos de las operaciones de estereotipo <<entry>>. El modelo gestiona automáticamente la sobrecarga en los procesadores debida a los cambios de contexto entre tareas de un mismo procesador.

Modela los bloqueos que introduce el acceso a objetos protegidos: Los objetos de la clase Protected se emplean para modelar los objetos protegidos Ada. El modelo incluye: la exclusión mutua con que se ejecutan las operaciones de su interfaz, la evaluación de la condición de guarda de sus Entry, el cambio de prioridad que introduce el uso del protocolo de techo de prioridad y su posible suspensión hasta que se alcance el estado en el que la condición de guarda se satisface. Aunque la metodología que se propone no es capaz de modelar todas las posibilidades de sincronización que se pueden codificar el emplear condiciones de guarda en las entry de un objeto protegido, sí que permite describir los mecanismos básicos de sincronización que son propios de las aplicaciones de tiempo real. Así, mecanismos de sincronización basados en objetos protegidos, tales como recepción de interrupciones hardware, activación periódica o asíncrona de tareas, espera a un grupo de eventos, o colas de mensajes, entre otros, pueden ser modelados cuantitativa y fielmente.

Modela la comunicación de tiempo real entre particiones Ada distribuidas: El modelo soporta de forma implícita el acceso local y remoto a los procedimientos APC y RPC de una *Remote Call Interface* tal como se describe en el Anexo E del estándar Ada. En la declaración de una RCI, el modelo incluye la información necesaria para que cuando un procedimiento sea invocado de forma remota, el *marshalling*, la transferencia de los mensajes por la red, el *unmarshalling* y su gestión por el *dispatcher* remoto, puedan ser incluidos de forma automática por la herramienta.

5 Ejemplo de aplicación de la metodología

Se muestra la utilización de ADA_MAST para el modelado y análisis de tiempo real de un sistema distribuido que controla una Máquina Herramienta Teleoperada (TMT por sus siglas en inglés). La plataforma del sistema está formada por dos procesadores comunicados mediante un bus CAN. El procesador **Station** hace de estación de teleoperación y aloja una aplicación tipo GUI a través de la cual el operador gestiona las tareas de la herramienta y en la que se muestra el estado del sistema; tiene asociado además un botón de emergencia que permite detener en seco la operación de la herramienta. **Controller** es un procesador empotrado que contiene el controlador de los servos de la máquina herramienta y la instrumentación asociada y que reporta periódicamente el estado de la herramienta a la estación de teleoperación.

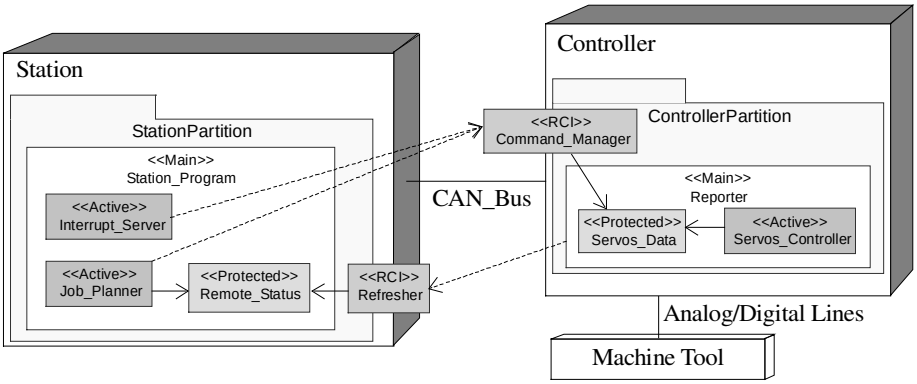


Fig. 6. Diagrama de despliegue de la Máquina Herramienta Teleoperada

5.1 Diseño lógico de la aplicación

Cada procesador tiene una partición Ada que ofrece al otro una interfaz de acceso remoto RCI. La figura 6 muestra los principales componentes y sus relaciones.

El programa principal de la partición Controller colecta el estado de la máquina herramienta cada 100ms e invoca en la otra partición el procedimiento remoto que actualiza su copia local del mismo. El objeto activo Servos_Controller que comanda la máquina herramienta tiene una tarea activada por un *timer* periódico cada 5ms. La interfaz de acceso remoto Command_Manager ofrece dos procedimientos, uno para procesar los encargos que se ordenan desde la estación y el otro para atender el eventual comando de parada de emergencia. Todos los componentes del procesador Controller comparten información mediante el objeto protegido Servos_Data.

El programa principal de la partición Station es una típica interfaz de usuario gráfica que comparte el acceso al objeto protegido Remote_Status con la interfaz de acceso remoto Refresher y dos objetos activos: la tarea Job_Planner que monitoriza y gestiona de manera periódica los encargos que se hacen a la máquina herramienta y la tarea Interrupt_Server que atiende a la interrupción hardware del botón de parada de

emergencia. La interfaz Refresher exporta un procedimiento para la actualización del estado de la máquina herramienta en la estación desde la partición remota.

Las plataformas Ada con las que se implementa el software de ambos procesadores deben contar no sólo con las librerías necesarias para la comunicación entre ambos, sino también con una implementación del *package* System.RCP que se atenga a las recomendaciones sobre el anexo E necesarias para ofrecer garantías de tiempo real.

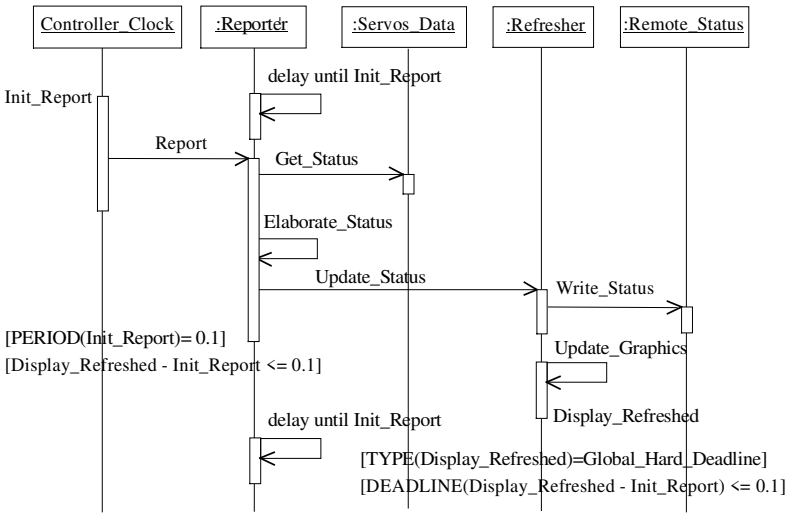


Fig. 7. Diagrama de secuencia que describe la transacción Report_Process

Este ejemplo tiene un único modo de operación y por tanto se define una sola Real_Time_Situation en la que todos los eventos, bien sean de temporizadores o externos, disparan hasta cuatro transacciones independientes con requisitos de tiempo real, y comparten tanto los recursos de procesamiento (Station, Controller, y CAN_Bus) como los objetos protegidos mencionados. Estas transacciones son:

- Control_Servos_Process: ejecuta el procedimiento Control_Servos con un periodo y plazo de finalización (*deadline*) de 5 ms.
- Report_Process: transfiere el estado de sensores y servos de Controller a Station para refrescar el *display* con un periodo y *deadline* de 100 ms.
- Drive_Job_Process: se activa por el *timer* cada segundo para revisar el estado de los encargos en curso y enviar los pendientes a la máquina herramienta.
- Do_Halt_Process: es activada de manera totalmente esporádica por el operador mediante el botón de parada de emergencia y se presume que el tiempo mínimo entre comandos sucesivos sea de 5 s. y el plazo de atención sea de 5 ms.

La figura 7 muestra la descripción funcional de la transacción Report_Process mediante un diagrama de secuencia.

5.2 Vista de tiempo real de la Máquina Herramienta Teleoperada

Se describen aquí las tres secciones de la vista UML de tiempo real de este ejemplo:

5.2.1 Modelo de la plataforma

Describe la capacidad de procesamiento y de conexión entre los tres recursos de procesamiento del sistema: los procesadores Station y Controller y la red CAN_Bus.

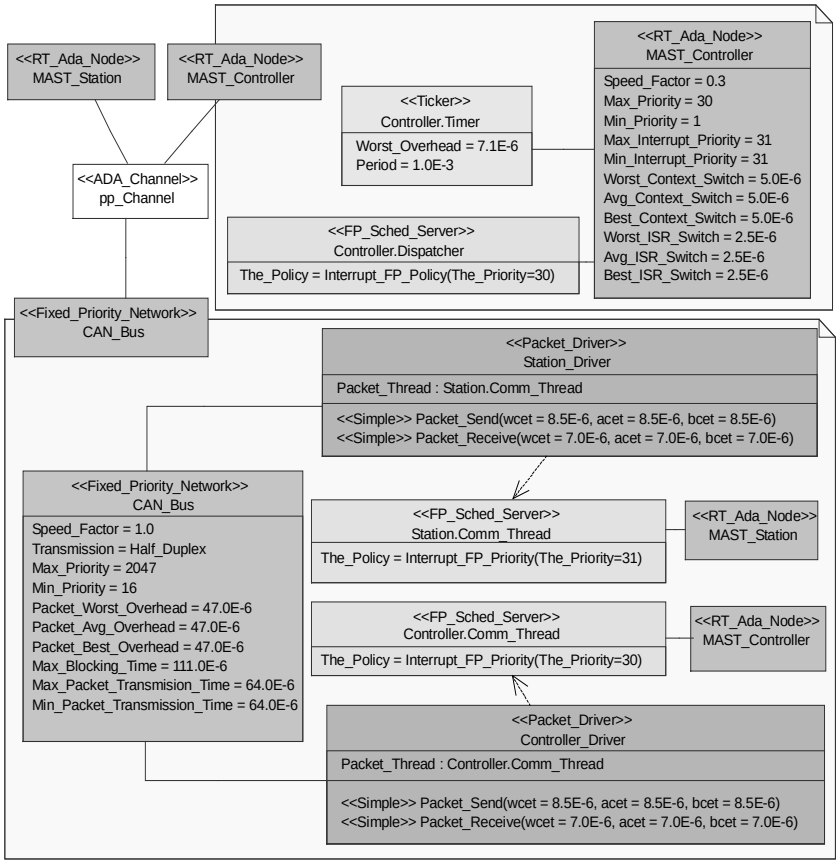


Fig. 8. Visión parcial del modelo de la plataforma de la Máquina Herramienta Teleoperada.

En la parte superior izquierda de la figura 8 se muestra el modelo de alto nivel y la conectividad que hay entre los elementos de la plataforma y a su derecha y debajo se detallan los modelos de la partición Controller y de la red CAN_Bus. La capacidad de procesamiento y los atributos del procesador están modelados por MAST_Controller de la clase RT_Ada_Node. Se trata de un procesador empotrado que ejecuta la partición Controller sobre un núcleo de tiempo real mínimo MaRTE OS [10], el cual emplea un *timer* del tipo *ticker* para temporizar las tareas que tiene asignadas. En esta parte del modelo se hacen explícitos sólo los threads de comunicaciones (*drivers*) y los propios del sistema operativo. La política de planificación y la prioridad asignadas al thread dispatcher de la partición, se especifican mediante el atributo The_policy, (del tipo Interrupt_FP_Policy en este caso) y su argumento The_Priority.

Los valores dados a los atributos propios del procesador Controller son:

Max_Priority= 30 Min_Priority= 1	Rango de prioridades admitido por el compilador Ada y MaRTE OS.
Max_Interrupt_Priority= 31 Min_Interrupt_Priority=31	Rango de prioridades admitido para manejadores de interrupciones hardware.
Worst_Context_Switch=5.0E-6 Avg_Context_Switch=5.0E-6 Best_Context_Switch=5.0E-6	Tiempo estimado de cambio de contexto entre threads de aplicación
Worst_ISR_Switch= 2.5E-6 Avg_ISR_Switch= 2.5E-6 Best_ISR_Switch= 2.5 E-6	Tiempo estimado de cambio de contexto entre un thread de aplicación y una rutina de atención a interrupciones o viceversa.
Speed_Factor= 0.3	El procesador Controller tiene un 30% de la capacidad de procesamiento del procesador de referencia del sistema.

La red CAN_Bus es un canal de tipo *half_duplex* orientado a paquetes, cada uno tiene una cabecera de 47 bits y un campo de datos de 8 bytes. La velocidad de transferencia del bus es de 1 Mbit/s. La transferencia de los mensajes es priorizada, esto es, no se transfiere ningún paquete de un mensaje de una prioridad dada, si aún quedan pendientes de transferencia paquetes de un mensaje de mayor prioridad.

El modelo del canal de comunicación se describe mediante un componente del tipo Fixed_Priority_Network y los Packet_Driver que corren en los procesadores en que están desplegadas las particiones que acceden a la red. La instancia CAN_Bus de la clase Fixed_Priority_Network describe la capacidad de transferencia por la red y los atributos que tiene y que modelan su comportamiento de tiempo real son:

Max_Priority= 2047 Min_Priority= 16	Rango de prioridades admitida para los mensajes sobre un bus CAN.
Packet_Worst_Overhead=47.0E-6 Packet_Avg_Overhead=47.0E-6 Packet_Best_Overhead=47.0E-6	Sobrecarga debida a los bits del formato de trama de mensaje del bus CAN que no son de datos (47 bits por paquete).
Transmission= Half_Duplex	El modo de transmisión del bus CAN es *half-duplex*.
Max_Blocking= 111.0E-6	Tiempo de bloqueo de peor caso debido a un paquete.
Max_Packet_Transmission_Time= 6.4E-5 Min_Packet_Transmission_Time= 6.4E-5	Tiempo para transmitir el campo de datos de un paquete en este caso de una sola trama (8 bytes/packet).
Speed_Factor= 1.0	Sea esta la red de referencia del sistema (1 Mbit/s).

5.2.2 Modelo de tiempo real de los componentes lógicos

Describe el comportamiento temporal de todos los módulos del diseño lógico que podrían afectar la respuesta de tiempo real del sistema. En la figura 9 se muestra el modelo del programa principal de la partición Controller, MAST_Reporter. Se trata de una instancia de la clase Main del metamodelo, que se emplea como contenedora y tarea periódica a la vez. Declara como suyos los objetos, The_Data y The_Controller, mediante atributos de sus correspondientes tipos. La política de planificación y la prioridad del thread interno se declaran también como atributos. La descripción de la operación compuesta Report se hace en un diagrama de actividad agregado y sigue la sintaxis propuesta por UML_MAST [9]. Los argumentos de la operación simple Elaborate_Report, dan valor a los tiempos de peor, medio y mejor caso esperados para el tiempo de ejecución de la operación. En la descripción del tipo de objeto protegido MAST_Servos_Data y del tipo de tarea MAST_Servos_Controller (asignados después a The_Data y a The_Controller), los argumentos de sus operaciones han sido

omitidos tan sólo para dar simplicidad y claridad a la figura. Obsérvese como el argumento The_rc de la operación compuesta Report se emplea internamente en su descripción para invocar el procedimiento Update_Status de la interfaz Refresher (que es del tipo APC), la cual a su vez ha sido declarada con un atributo <<ref>> y es accesible por tanto dentro de MAST_Reporter bajo el nombre Remote_Refresher.

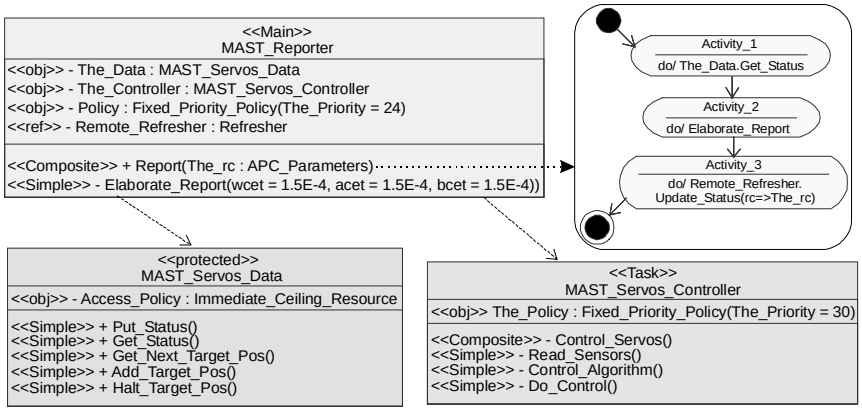


Fig. 9. Extracto de la descripción del componente MAST_Reporter

5.2.3 Modelo de las situaciones de tiempo real

La situación de tiempo real que se ha de analizar comprende la formulación de las cuatro transacciones descritas en el apartado 5.1. En la figura 10 se muestra el modelo de la transacción Report_Process, que fue descrita funcionalmente en la figura 7.

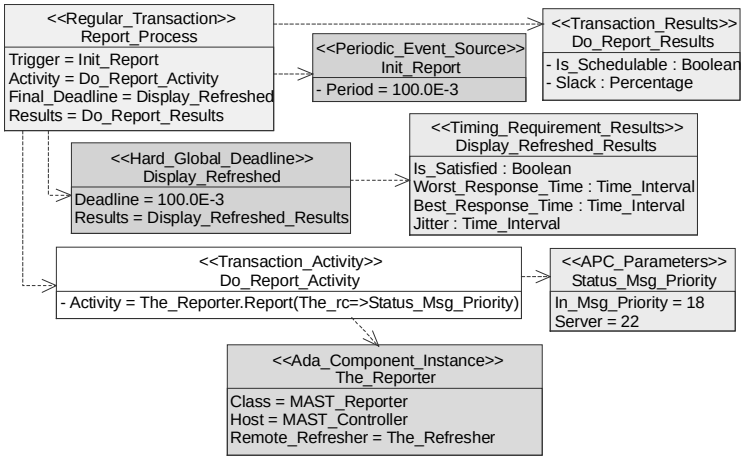


Fig. 10. Transacción Report_Process

Para analizar el sistema se requiere tener declarados en la situación de tiempo real los cuatro componentes de primer nivel que modelan la aplicación y que corresponden a los dos programas principales: The_Reporter y The_Station_Program y a las dos interfaces exportadas: The_Command_Manager y The_Refresher.

La transacción Report_Process se declara como una instancia de la clase Regular_Transaction, su atributo Trigger referencia el Periodic_Event_Source Init_Report y Final_Deadline al Hard_Global_Deadline Display_Refreshed la actividad a ser iniciada por la transacción es la operación Report del objeto The_reporter. El atributo Period de Init_Report indica que se recibe un evento cada 100 ms. El atributo Deadline de Display_Refreshed indica que el plazo para alcanzar el estado Display_Refreshed que aparece en la descripción de Do_Report_Activity es de 100 ms. contados desde la generación del evento de disparo (Init_Report). La descripción detallada de la secuencia de actividades que conlleva la transacción se obtiene sustituyendo de manera recursiva todos los parámetros y modelos de actividad de las operaciones tal como se les declara en el modelo lógico, con los valores instanciados a partir de los objetos que se declaran en la Situación de tiempo real. Las *do_actions* de las actividades invocan operaciones y los *swim lanes* representan los Scheduling_Servers (threads) en los que se ejecutan.

5.3 Análisis de tiempo real y diseño de la planificabilidad del sistema

Utilizando las herramientas del entorno MAST se ha analizado el ejemplo que se presenta a fin de obtener los tiempos de bloqueo y techos de prioridad de los recursos, la asignación óptima de prioridades y la holgura disponible tanto para el sistema como para cada transacción independientemente. Se ha empleado el método de análisis de planificabilidad basado en offsets por ser el menos pesimista [11][12]. En la tabla 1 se muestra un resumen de los resultados de planificabilidad obtenidos para cada transacción, pudiendo compararse el tiempo de respuesta de peor caso de cada una con su correspondiente plazo. Las prioridades asignadas a las tareas han sido calculadas con el algoritmo HOPA integrado en MAST.

Table 1. Resultados del análisis para las cuatro transacciones de la situación de tiempo real

Transaction/Event	Slack	Worst response	Deadline
Control_Servos_Process	19.53%		
End_Control_Servos		3.833ms	5 ms
Report_Process	254.69%		
Display_Refreshed		34.156ms	100 ms
Drive_Job_Process	28.13%		
Command_Programmed		177.528ms	1000 ms
Do_Halt_Process	25.00%		
Halted		4.553ms	5 ms

En la tabla 1 se muestran también las holguras de cada transacción. La holgura (*Slack*) es el porcentaje en el que todas las operaciones involucradas pueden ser incrementadas sin hacer el sistema no planificable o decrementadas para que lo sea si es negativo. Como se puede apreciar el porcentaje en el que se puede aumentar los tiempos de ejecución de las operaciones de la transacción Report_Process

manteniendo la planificabilidad del sistema es de 254.69%. Mientras que desde el punto de vista global el tiempo de ejecución de todas las operaciones del sistema tan sólo podría crecer un 2.34%, ya que es éste el valor de la holgura calculada para el sistema.

6 Conclusión

En este trabajo se ha presentado una metodología para modelar el comportamiento de tiempo real de una aplicación codificada en Ada, con el suficiente nivel de detalle como para que herramientas tales como el análisis de planificabilidad, la asignación optima de prioridades, el cálculo de holguras, etc. sean aplicables.

Su principal característica es que permite modelar componentes Ada complejos (*package*, *tagged class*, *protected objects*, *tasks*, etc), de forma independiente al uso que de ella se haga en una aplicación concreta, lo que sirve como base de una metodología de diseño de sistemas de tiempo real basada en componentes Ada reusables.

La metodología libera al diseñador de la aplicación de tener que modelar los mecanismos internos del sistema operativo o del ejecutivo de tiempo real que soporta la aplicación Ada. Así, el modelado de los cambios de contexto entre tareas, de las tareas de background de gestión de las comunicaciones o de los *timers*, los *mutexes* de acceso a los objetos protegidos, o los mecanismo de acceso a interfaces remotas (RCI), son soportados a partir de un modelo de la plataforma independiente de la aplicación o del código Ada de los componentes.

El poder de modelado de la metodología que se presenta es capaz de cubrir la mayor parte de los patrones o estructuras software que son de uso extendido en el desarrollo de la mayoría de aplicaciones Ada que son analizables. Desde luego el modelado de componentes que implican mecanismos complejos de sincronización basados en su estado interno (que en Ada se implementan mediante condiciones de guarda de *entries* en tareas o en objetos protegidos) y que conduce a situaciones no analizables, no está de momento resuelto con carácter general, y en tales casos no es aplicable de manera directa la metodología propuesta. Sin embargo justamente por su no analizabilidad estas estructuras no son frecuentes en sistemas de tiempo real

La metodología que se ha presentado está actualmente siendo implementada como parte de las herramientas del entorno UML-MAST y al igual que las de simulación y la de modelado basado en componentes de tiempo real, constituye una de nuestras líneas principales de trabajo futuro. La descripción detallada de estas herramientas y metodologías y las implementaciones que de ellas estén disponibles se pueden encontrar en: http://mast.unican.es

Bibliografía

[1] S. Tucker Taft, and R.A. Duff (Eds.) “Ada 95 Reference Manual. Language and Standard Libraries”. International Standard ISO/IEC 8652:1995(E), in Lecture Notes on Computer Science, Vol. 1246, Springer, 1997.

[2]L. Pautet and S. Tardieu, “Inside the Distributed Systems Annex”, Intl. Conf. on Reliable Software Technologies, Ada-Europe’98, Uppsala, Sweden, in LNCS 1411, Springer, pp. 65-77, June 1998.

[3] L. Pautet and S. Tardieu: “GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems. In Proceedings of the 3rd IEEE”. International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00), Newport Beach, California, USA, March 2000.

[4]Luís Miguel Pinho, "Distributed and Real-Time: Session summary", 10th.Intl. Real-Time Ada Workshop, IRTAW'01, Ávila,Spain, in Ada Letters, Vol. XXI, Number 1, March 2001.

[5] Ada-Core Technologies, Ada 95 GNAT Pro Development Environment. http:// www.gnat.com/

[6] J.J. Gutiérrez García, and M. González Harbour: “Prioritizing Remote Procedure Calls in Ada Distributed Systems”. 9th International Real-Time Ada Workshop, ACM Ada Letters, XIX, 2, pp. 67-72, June 1999.

[7] J.J. Gutiérrez García, and M. González Harbour: “Towards a Real-Time Distributed Systems Annex in Ada”. 10th International Real-Time Ada Workshop, ACM Ada Letters, XXI, 1, pp. 62-66, March 2001.

[8] M. González Harbour, J.J. Gutiérrez, J.C. Palencia and J.M. Drake: “MAST: Modeling and Analysis Suite for Real-Time Applications” Proceedings of the Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.

[9] J.L. Medina, M. González Harbour, and J.M. Drake: “MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems” RTSS’01, London, December, 2001

[10] M. González Harbour, and M. Aldea: “MaRTE OS: An Ada kernel for Real-Time Embedded Applications". Int. Conf. on Reliable Software Technologies, Ada_Europe'01. Leuven, May, 2001

[11] J.C. Palencia, and M. González Harbour, “Schedulability Analysis for Tasks with Static and Dynamic Offsets”. Proc. of the 19th IEEE Real-Time Systems Symposium, 1998.

[12] J.C. Palencia, and M. González Harbour, “Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems”. Proceedings of the 20th IEEE Real-Time Systems Symposium, 1999.

[13] J.M. Drake, J.L. Medina y M. González Harbour: ”Entorno para el Diseño de Sistemas Basados en Componentes de Tiempo Real”, X Jornadas de Concurrencia, Jaca, Junio, 2002

[14] Selic B., Moore A., Woodside M., Watson B., Bjorkander M., Gerhartdt M., y Petriu D.: "Response to the OMG RFP for Schedulability, Performance and Time” OMG document ad/2001-06-14. Junio, 2001.

A Low-Latency Non-Blocking Commit Service*

R. Jiménez-Peris[1], M. Patiño-Martínez[1], G. Alonso[2], and S. Arévalo[3]

[1] Technical University of Madrid (UPM), Facultad de Informática,
E-28660 Boadilla del Monte, Madrid, Spain, {rjimenez, mpatino}@fi.upm.es
[2] Swiss Federal Institute of Technology (ETHZ), Department of Computer Science, CLU E1,
ETH-Zentrum, CH-8092 Zürich, Switzerland, alonso@inf.ethz.ch
[3] Universidad Rey Juan Carlos, Escuela de Ciencias Experimentales, Móstoles, Madrid, Spain,
s.arevalo@escet.urjc.es

Abstract. Atomic commitment is one of the key functionalities of modern information systems. Conventional distributed databases, transaction processing monitors, or distributed object platforms are examples of complex systems built around atomic commitment. The vast majority of such products implement atomic commitment using some variation of 2 Phase Commit (2PC) although 2PC may block under certain conditions. The alternative would be to use non-blocking protocols but these are seen as too heavy and slow. In this paper we propose a non-blocking distributed commit protocol that exhibits the same latency as 2PC. The protocol combines several ideas (optimism and replication) to implement a scalable solution that can be used in a wide range of applications.

1 Introduction

Atomic commitment (AC) protocols are used to implement atomic transactions. *Two-phase commit* (2PC) [8] is the most widely used AC protocol although its blocking behavior is well known. There are also non-blocking protocols but they have an inherent higher cost [6, 14] usually translated in either a explicit extra round of messages (*3 phase commit* (3PC) [21, 22, 13]) or an implicit one (when using uniform multicast [1]).

The reason why 2PC is the standard protocol for atomic commitment is that transactional systems pay as much attention to performance as they do to consistency. For instance, most systems summarily abort those transactions that have not committed after a given period of time so that they do not keep resources locked. Existing non-blocking protocols resolve the consistency problem by increasing the latency and, therefore, are not practical. A realistic non-blocking alternative to 2PC needs to consider both consistency and transaction latency. Ideally, the non-blocking protocol should have the same latency as 2PC. Our goal is to implement such a protocol by addressing the three main sources of delay in atomic commitment: message overhead, forced writes to the log, and the *convoy effect* caused by transactions waiting for other transactions to commit.

To obtain non-blocking behavior, it is enough for the coordinator to use a virtual synchronous uniform multicast protocol to propagate the outcome of the transaction

* This research has been partially funded by the Spanish National Research Council CICYT under grant TIC98-1032-C03-01.

[1]. This guarantees that either all or none of participants know about the fate of the transaction. Uniformity [12] ensures the property holds for any participant, even if it crashes during the multicast. Unfortunately, uniformity is very expensive in terms of the delay it introduces. In addition, since the delay depends on the size of the group, using uniformity seriously compromises the scalability of the protocol. To solve this two limitations, we use two different strategies. First, to increase the scalability, uniform multicast is used only within a small group of processes (the *commit servers*) instead of using it among all participants in the protocol. The idea is to employ a hierarchical configuration where a small set of processes run the protocol on behalf of a larger set of participants. Second, to minimize the latency caused by uniformity, we resort to a novel technique based on optimistic delivery that overlaps the processing of the transactional commit with the uniform delivery of the multicast. The idea here is to hide the latency of multicast behind operations that need to be performed anyway. This is accomplished by processing messages in an optimistic manner and hoping that most decisions will be correct although in some cases transactions might need to be aborted. This approach builds upon recent work in optimistic multicast [18] and a more aggressive version of optimistic delivery proposed in the context of Postgres-R [15] and later used to provide high performance eager replication in clusters [17]. We use an optimistic uniform multicast that delivers messages in two steps. In the first step messages are delivered optimistically as soon as they are received. In the second step messages are delivered uniformly when they become stable. This optimistic uniform multicast is equivalent to a uniform multicast with safe indications [23].

Forced writes to the log are another source of inefficiencies in AC protocols. To guarantee correctness in case of failures, participants must flush to disk a log entry before sending their vote. This log entry contains all the information needed by a participant to recall its own actions in the event of a crash. The coordinator is also required to flush the outcome of the protocol before communicating the decision to the participants (this log entry can be skipped by using the so called *presume commit* or *presume abort* protocols [16]). Flushing log records adds to the overall latency as messages cannot be sent or responded to before writing to the log. In the protocol we propose, this delay is reduced by allowing sites to send messages instead of flushing log records. The idea is to use the main memory of a replicated group (the commit servers mentioned above) as stable memory instead of using a mirrored log with careful writes.

Finally, to minimize the waiting time of transactions, in our protocol locks are released optimistically. The idea is that a transaction can be optimistically committed pending the confirmation provided by the uniform multicast. By optimistically committing the transaction, other transactions can proceed although they risk a rollback if the transaction that was optimistically committed ended up aborting. In our protocol, the optimistic commit is performed in such a way that aborts are confined to a single level. In addition, transactions are only optimistically committed when all their participants have voted affirmatively, thereby greatly reducing the risk of having to abort the transaction. This contrasts with other optimistic commit protocols, e.g., [11], where transactions that must abort (because one or more participants voted abort) can be optimistically committed although they will rollback anyway producing unnecessary cascading aborts.

With these properties the protocol we propose satisfactorily addresses all design concerns related to non-blocking AC and can thus become an important contribution to future distributed applications. The paper is organized as follows, Section 2 describes the system model. Section 3 and 4 present the commit algorithm and its correctness. Section 5 concludes the paper.

2 Model

2.1 Communication Model

The system consists of a set of fail-crash processes connected through reliable channels. Communication is asynchronous and by exchanging messages. A failed process can later recover with its permanent storage intact and re-join the system. Failures are detected using a (possibly unreliable) failure detector[1] [5].

A virtual synchronous multicast service [3, 4, 19] is used. This service delivers multicast messages and views. Views indicate which processes are perceived as up and connected. We assume a virtual synchrony with the following properties: (1) *Strong virtual synchrony* [7] or sending view delivery [23] that ensures that messages are delivered in the same view they were sent; (2) *Majority or primary component views* that ensure that only members within a majority view can progress, while the rest of the members block until joining again the majority view; (3) *Liveness*, when a member fails or it is partitioned from the majority view, a view excluding the failed member will be eventually delivered.

The protocol uses two different multicast primitives [12, 20]: reliable (*rel-multicast*) and uniform multicasts (*uni-multicast*). Three primitives define optimistic uniform reliable multicast[2]: *Uni-multicast(m, g)* multicasts message m to a group g. *Opt-deliver(m)* delivers m reliably to the application. *Uni-deliver(m)* delivers m to the application uniformly. We say that a process is $v_i - correct$ in a given view v_i if it does not fail in v_i and if v_{i+1} exists, it transits to it. The rel- and uni-multicasts preserve the following properties, where m is a message, g a group of processes and v_i a view within this group:

OM-Validity: If a correct process rel or uni-multicast m to g in v_i, m will be eventually opt-delivered by every v_i-correct process.

OM-Agreement: If a v_i-correct process opt-delivers m in v_i, every v_i-correct process will eventually opt-deliver m.

OM-Integrity: Any message is opt and uni-delivered by a process at most once. A message is opt-delivered only if it has been previously multicast.

Uni-multicast additionally fulfills the following properties:

[1] The failure detector must allow the implementation of the virtual synchrony model described below (e.g., the one proposed in [19]) and the non-blocking atomic commitment (e.g., the one in [10]).

[2] Senders are not required to belong to the target group.

OM-Uniform-Agreement: If a majority of v_i-correct processes opt-deliver m, they will eventually uni-deliver m. If a process uni-delivers m in v_i, every v_i-correct process will eventually uni-deliver m.

OM-Uniform-Integrity: A message is uni-delivered in v_i only if it was previously opt-delivered by a majority of processes in v_i.

2.2 Transaction Model

Clients interact with the database by issuing transactions. A transaction is a partially ordered set of *read* and *write* operations followed by either a *commit* or an *abort* operation. The decision whether to commit or abort is made after executing an optimistic atomic commitment protocol. The protocol can decide (1) to immediately abort the transaction, (2) to perform an *optimistic commit*, or (3) decide to commit or abort.

We assume a distributed database with n sites. Each site i has a *transaction manager* process TM_i. When a client submits a transaction to the system, it chooses a site as its *local* site. The local TM_i decides which other sites should get involved in processing the transaction and initiates the commitment protocol.

We assume the database uses standard mechanisms like *strict 2 phase locking* (2PL) to enforce *serializability* [2]. The only change over known protocols is introduced during the commit phase of a transaction. When a transaction *t* is optimistically committed, all its write locks are changed to *opt* locks and all its read locks are released[3]. *Opt* locks are compatible with all other types of locks. That is, other transactions are allowed to set compatible locks on data held under an *opt* lock. Such transactions are said to be *on-hold*, while the rest of transactions are said to be on *normal* status. When the outcome of t is finally determined, its *opt* locks are released and all transactions that were on-hold due to these *opt* locks are returned to their normal state. A transaction that is on-hold cannot enter the commit phase until it returns to the normal state.

2.3 System configuration

For the purposes of this paper, we will assume there are two disjoint groups of processes in the system. The first will conform the distributed database and will be referred as the *transaction managers* or TM group ($TM = \{TM_1, ..., TM_n\}$). Sites in this group are responsible for executing transactions and for triggering the atomic commitment protocol. By participants in the protocol, we mean processes in this group. The second group, *commit server* or CS group ($CS = \{C_1, ..., C_n\}$) is a set of replicated processes devoted to perform the AC protocol. We assume that in any two consecutive views, there is a process that transits from the old view to the new one[4].

[3] Once a transaction concludes, all its read locks can be released without compromising correctness independently of whether the transaction commits or aborts [2].

[4] It might seem a strong assumption for safety that at least one server must survive between views. However, this assumption is no stronger than the usual one that assumes that the log is never lost. The strength of any of the assumptions depends on the probability of the corresponding catastrophic failures.

2.4 Problem definition

A non-blocking AC protocol should satisfy: (1) NBAC-Uniform validity, a transaction is (opt) committed only if all the participants voted yes; (2) NBAC-Uniform-Agreement, no two participants decide differently; (3) NBAC-Termination, if there is a time after which there is a majority view sequence in the CS group that permanently contains at least a correct process, then the protocol terminates; (4) NBAC-Non-Triviality [9], if all participants voted yes, and there no failures or false suspicions, then commit is decided.

3 A Low Latency Commit Algorithm

3.1 Protocol Overview

The AC protocol starts when a client requests to commit a transaction. The commit request arrives at a transaction manager, TM_i, which then starts the protocol. The protocol involves several rounds of messages in two phases:

First phase

1. Upon delivering the commit request, TM_i multicasts a reliable *prepare to commit* message to the TM group. This message contains the transaction identifier (*tid*) to be committed and the number of participants involved (the number of TMs contacted during the execution of the transaction).
2. Upon delivering the *prepare to commit* message, each participant uni-multicasts its *vote* and the number of participants to the CS group. If a participant has not yet written the corresponding entries to its local log when the *prepare to commit* message arrives, it sends the log entry in addition to its vote without waiting to write to the log. After the message has been sent, it then writes the log entry to its local disk.

Second phase

1. Upon *opt-delivering* a *vote* message, the processes of the commit server decide who will act as *proxy coordinator* for the protocol based on the tid of the transaction and the current view. Assume this site is C_i. The rest of the processes in the CS group act as backup in case C_i fails. If a no vote is opt-delivered, the transaction is aborted immediately and an *abort* message is reliable multicast to the TM group. If all votes are yes, as soon as the last vote is opt-delivered at C_i, C_i sends a reliable multicast with an *opt-commit* message to the TM group.
2. Upon delivering an *abort* message, a participant aborts the transaction. Upon delivering an *opt-commit* message, the participant changes the transaction locks to *opt* mode.
3. If all votes are affirmative, when they have been *uni-delivered* at C_i, C_i reliable multicasts to the TM group a *commit* message.

4. When a participant delivers a *commit* or *abort* message, it releases all locks (both opt and non-opt) held by the transaction and return the corresponding transactions that were on hold to their normal state.
5. If all the votes are affirmative, the coordinator opt-commits the transaction before being excluded from the majority view (before being able to commit the transaction), and one or more votes do not reach the majority view, the transaction will be aborted by the new coordinator.

This protocol reduces the latency of the non-blocking commit in several ways. First, at no point in time in the protocol must a site wait to write a log entry to the disk before reacting to a message. The CS group acts as stable storage for both the participants (sites at the TM which could not yet write their vote and other transaction information to disk when the *prepare to commit* vote arrives) and the CS group itself (the coordinator does not need to write an entry to the log before sending the *opt-commit* message). Second, the coordinator in the CS group provides an outcome without waiting for the *vote* messages to be uniform. This reduces the overhead of uniform multicast as it overlaps its cost with that of committing the transaction.

3.2 The Protocol

The protocol uses the CS group to run the atomic commitment. The processes in the TM group[5] only act as participants and the CS group acts as coordinator. We use two tables, *trans_tab* and *vote_tab*, to store information in main memory about the state of a transaction and the decision of each participant regarding a given transaction at each CS_i. We also use a number of functions to change and access the values of the attributes in these tables. *Trans_tab* contains the attributes *tid* (the transaction's identifier), *n_participants* (number of participants in that transaction, all sites in the TM group), *timestamp* (of the first vote for timeout purposes), *coordinator* (id of the coordinator site in the CS group; this attribute is initially set with the function store_trans and updated with the function store_coordinator), and *outcome* (the state of the transaction; initially it is undecided, the state can be changed to aborted, opt-committed or committed by invoking the function *store_outcome*). *Vote_tab* contains the attributes *tid* (the transaction's identifier), *participant_id* (site emitting the vote, which must be a site in the TM group), *vote* (the actual vote), *vote_status* (initially optimistic, when set with the function store_opt_vote, and later definitive, when set with the function store_def_vote), and *log* (any log entry the participant may have sent with the vote). There are additional functions to consult the attributes associated to each tid in the *trans_tab*. These functions are denoted with the same name as the attribute but starting with capital letter (e.g., Timestamp). There are also functions to consult the *vote_tab*: Log (to obtain the log sent by a participant), *N_opt_yes_votes* (number of yes votes delivered optimistically for a particular transaction), *N_def_yes_votes* (similarly for uni-delivered yes votes). An additional function, *Coordinator*, is used to obtain the id of the coordinator of a transaction given its tid and the current view.

[5] For simplicity, messages are multicast to all TM processes. Processes for which the message is not relevant just discard it.

TM Group actions:

TM.A Upon **delivering Prepare**(tid):
if prepared in advance then
uni-multicast(CS, Vote(tid, n_participants, my_id, vote, empty))
else
uni-multicast(CS, Vote(tid, n_participants, my_id, vote, log_record))
end if

TM.B Upon **delivering Opt-commit**(tid):
Change transaction tid locks to opt-mode

TM.C Upon **delivering Commit/Abort**(tid):
Commit/Abort the transaction and release transaction tid locks
Change the corresponding on-hold transactions to normal status

CS Group actions:

CS.A Upon **opt-delivering Vote**(tid, n_participants, participant_id, vote, log):
store_opt_vote(vote_tab, tid, participant_id, vote, log)
– the transaction outcome is still undecided
if Outcome(trans_tab, tid) = undecided then
if vote = no then
if Coordinator(current_view, tid) = my_id then
rel_multicast(TM, Abort(tid))
end if
store_outcome(trans_tab, tid, aborted)
else – *vote = yes*
if N_opt_yes_votes(vote_tab, tid) = 1 then – *it is the first vote*
timestamp = current_time
if Coordinator(current_view, tid) = my_id then
set_up_timer(tid, timestamp+waiting_time)
end if
store_trans(trans_tab, tid, n_participants, timestamp)
end if
if N_opt_yes_votes(vote_tab, tid) = n_participants(trans_tab, tid) then – *all voted yes*
store_outcome(trans_tab, tid, opt-committed)
if Coordinator(current_view, tid) = my_id then
disable_timer(tid)
rel_multicast(TM, Opt-commit(tid))
end if
end if
end if

CS.B Upon **uni-delivering Vote**(tid, n_participants, participant_id, vote, log):
store_def_vote(trans_tab, tid, participant_id)
if (N_def_yes_votes(vote_tab, tid) = n_participants(trans_tab, tid))
and (Outcome(trans_tab, tid) $\neq$ abort) then
store_outcome(trans_tab, tid, committed)
if Coordinator(current_view, tid) = my_id then
rel_multicast(TM, Commit(tid))
end if
end if

CS.C Upon **expiring Timer**(tid):
store_outcome(trans_tab, tid, aborted)
uni_multicast(CS, Timeout(tid))

CS.D Upon **uni-delivering Timeout**(tid):
store_outcome(trans_tab, tid, aborted)
if Coordinator(current_view, tid) = my_id then
rel_multicast(TM, Abort(tid))
end if

```
CS.E  Upon delivering ViewChange(v_i):
  current_view = v_i
  – State synchronization with new members
  if my_id is the lowest in v_i that belonged to v_i-1 then
    for every C_i ∈ v_i do
      if C_i ∉ v_i-1 then
        send(C_i, State(trans_tab, vote_tab))
      end if
    end for
  elsif my_id ∉ v_i-1 then I am a new member
    receive(State(trans_tab, vote_tab))
  end if
  – Assignment of new coordinators in v_i
  for each tid ∈ trans_tab do
    if Coordinator(v_i, tid) = my_id then
      if Outcome(trans_tab, tid) = committed then
        rel_multicast(TM, Commit(tid))
      elsif Outcome(trans_tab, tid) = aborted then
        rel_multicast(TM, Abort(tid))
      else
        set_up_timer(Timestamp(trans_tab, tid) + waiting_time)
      end if
    end if
  end for
end if
```

Dealing with coordinator failures Since sites in the TM group only act as participants, failures in the TM group do not affect the protocol. In the CS group, all processes are replicas of each other. Strong virtual synchrony ensures that any pending message sent in the previous view is delivered before delivering a new view. Thus, when a process fails (or it is falsely suspected), a new view is eventually delivered to a majority of available connected CS processes. Once the new view is available, a working CS process takes over as coordinator for all the on-going commitment protocols coordinated by the failed process (CS.E). For each on-going transaction commit, the new coordinator checks the delivery time of the first vote and sets up a timer accordingly (CS.E). The actions taken by the new coordinator at this point in time depend on the protocol stage. If the transaction outcome is already known (all the votes have been opt-delivered at all CS members or a no vote message has been opt-delivered), the new coordinator multicasts the outcome to the participants (CS.E). If the outcome is undecided (i.e., all previously delivered votes were affirmative and there are pending votes), the protocol proceeds as normal and the new coordinator waits until all vote messages (or a no vote) have been *opt-delivered* (CS.A).

The problematic case is when the coordinator has decided to commit the transaction and then it is excluded from the view. It can be the case that the coordinator has had time to opt-commit the transaction, but not to commit it, and that there are missing votes in the majority view. In traditional 2PC, this situation is avoided by blocking. In our protocol, the blocking situation is avoided by the use of uniform multicast within the server group. The surviving sites can safely ignore the previous coordinator: due to uniformity, at worst, they will be aborting an opt-committed transaction, which does not violate consistency. A more accurate characterization of the rollback situation follows:

- All the votes are yes and have been opt-delivered at the coordinator.
- The coordinator successfully multicasts the opt-commit to the participants.
- The vote from at least one of the participants (e.g., p) is not uni-delivered.

– The coordinator and p become inoperative (e.g., due to a crash, a false suspicion, etc.) in such a way that the multicast does not reach any other member of the new primary view.

Although such a situation is possible, it is fair to say that it is extremely unlikely. Even during instability periods where false suspicions are frequent and many views are delivered, the odds for a message being opt-delivered only at the coordinator and both the coordinator and p become inoperative before the multicast of missing votes reaches any other CS member are very low. Additionally, this has to happen after the opt-commit is effective, as otherwise there would not be any rollback. Being such a rare situation, the amount of one-level aborts will be minimal even if the protocol is making optimistic decisions. It is also possible to enhance the protocol by switching optimism off during instability periods.

Bounding commit duration To guarantee the liveness of the protocol and to prevent unbounded resource contention it is necessary to limit the duration of the commit phase of a transaction. This limitation is enforced by setting a timer at the coordinator when it receives the first vote from a transaction (CS.A). The rest of the members timestamp the transaction with the current time when they opt-deliver the first vote. If all participant votes have reached the coordinator before the timer expires, the timer is disabled (CS.A). Otherwise, the coordinator decides to abort the transaction but it does not immediately multicast the abort decision to the TM group (CS.C). Instead, it uni-multicasts a *timeout* message to the CS group. When this message is uni-delivered at the coordinator, a message is sent to the participants with the abort decision (CS.D).

It could be the case that the coordinator multicast a timeout message and, before uni-delivering it, the missing votes are opt-delivered at the coordinator. In that case the transaction will be aborted (its outcome is not undecided when the vote is opt-delivered since in CS.C the outcome is set to abort when the timer expires). The rest of the CS members will also abort the transaction, no matter the order in which those messages are delivered. If the missing votes are delivered before the timeout message, the transaction outcome will be set to commit (CS.A) until the timeout message is uni-delivered (if so). Upon uni-delivery of the timeout message the outcome is changed to abort (CS.D). If the timeout message is uni-delivered before the last vote at a CS member, the transaction outcome will be initially set to abort (CS.D) and remain so (CS.A).

The coordinator can be excluded from the majority view during this process and a new coordinator will take over. If the new coordinator has uni-delivered the timeout message, the outcome of the transaction will be abort (CS.E). This will happen independently of whether the old coordinator sent the abort message to the TM group. If the new coordinator has not uni-delivered the timeout message before installing the new view, the failed coordinator did not uni-deliver it either (due to uniformity) and the new coordinator will not deliver that message (strong virtual synchrony). Therefore, the new coordinator will behave as a regular coordinator and will set the timer and wait for any pending vote (CS.E to set the timer, and CS.A in the event a vote arrives).

Despite the majority view approach, the protocol would not terminate if all the coordinators assigned to a transaction are excluded from the view before deciding the outcome. For instance, the CS group can transit perpetually between views $\{1,2,3,4\}$

and $\{1,2,3,5\}$, with processes 4 and 5 being the coordinators of a transaction t in each view. In this case, t will never commit. This scenario can be avoided by, whenever possible, choosing a coordinator that has not previously coordinated the transaction. It there is at least a correct process, this will guarantee that the outcome of t will be eventually decided, thereby, ensuring the liveness of the algorithm.

Maintaining consistency across partitions Although partitions always lead to blocking, our protocol maintains consistency even when partitions occur. That is, no replica decides differently on the outcome of a transaction even when the network partitions. Consistency is enforced by combining uniformity, strong virtual synchrony, and majority views. To see why, we will only consider partitions in the CS group. Partitions in the TM may lead to delays in the vote delivery (which may result in a transaction abort) and to delays in the propagation of the transaction outcome (thus, resulting in blocking during the partition). Partitions that leave the coordinator of a transaction in the majority partition of the CS group are not a problem, as the minority partition gets blocked (due to the majority view virtual synchrony). Since the transaction outcome is always decided after the uni-delivery of a message (either a vote or timeout message), uniformity guarantees that the decision will be taken by every process in the majority view. When the coordinator of a transaction is in a minority partition, undecided transactions cannot create problems as the coordinator, once in the minority partition, will block. When this happens, a new coordinator can make any decision regarding undecided transactions without compromising consistency. Only transactions whose outcome has been decided by the coordinator during the partition may lead to inconsistencies. There are four cases to consider:

- The coordinator optimistically commits a transaction when it opt-delivers the last vote (and all votes have been affirmative). Assume a partition leaves the coordinator in a minority partition. The new coordinator may (1) opt-deliver all the votes or (2) never deliver one or more votes. In the first case, it will opt-commit the transaction (CS.A) thereby agreeing with the old coordinator. In the second case, it will abort the transaction once the timer expires (CS.C). Since the transaction was only optimistically committed by the old coordinator, the new coordinator is free to decide to abort without violating consistency.
- If the old coordinator committed a transaction, the new coordinator will do the same. A transaction is committed when all the votes have been uni-delivered (CS.B). If the votes were uni-delivered at the old coordinator, all the processes in that view also uni-delivered them in that view (uniformity and strong virtual synchrony). Thus, the new coordinator will also commit the transaction (CS.E).
- The old coordinator opt-delivered a no vote and aborted the transaction. The new coordinator will either have delivered the no vote or timed out. In both cases the new coordinator will also abort the transaction (CS.A and CS.C, respectively) thereby agreeing with the old coordinator.
- The old coordinator timed out and aborted the transaction. The transaction will not be effectively aborted until the timeout message is uni-delivered. Uniformity guarantees that the timeout message, if uni-delivered, will be uni-delivered to both the old and the new coordinator, thereby preventing any inconsistency.

Replica recovery and partition merges In order, to maintain an appropriate level of availability, it is necessary to enable new (or recovered) replicas to join the CS group and to allow partitioned groups to merge again. When a new process joins the CS group, virtual synchrony guarantees that the new process will deliver all the messages delivered by the other replicas after installing the new view (and thus, after state synchronization). The installation of the new view will trigger state synchronization (CS.E). This involves sending from an old member of the group (one that transits from the previous view to the current one, that it is guaranteed to exist due to the majority view approach) a *State* message with the `vote_tab` and the `trans_tab` tables to the new member. Members from a minority partition that join a majority view will be treated as recovered members, that is, they will be sent the up-to-date tables from a process belonging to the previous majority view.

The state transfer and the assumption that at least a process from the previous view transits to the next view guarantees that a new member acting as coordinator will use up-to-date information, thereby ensuring the consistency of the protocol. The recovery of a participant has not been included in the algorithm due to its simplicity (upon recovery it will just ask to the CS group about the fate of some transactions).

4 Correctness

Lemma 1 (NBAC-Uniform-Validity). *A transaction (opt)commits only if all the participants voted yes.* □

Proof (lemma 1): (Opt) Commit is decided when the coordinator multicasts such message after the transaction has been recorded as (opt) committed in the *trans_tab* (CS.A). This can only happen when all participant votes have been (opt) uni-delivered and they are yes votes. □

Lemma 2 (NBAC-Uniform-Agreement). *No two CS members decide differently.* □

Proof (lemma 2): In the absence of failures the lemma is proved trivially, as only the coordinator decides about the outcome of the transaction. The rest of them just logs the information about the transaction in case they have to take over. The only way for two members to decide on the same transaction is that one is a coordinator of the transaction and then it is excluded from the majority view before deciding the outcome. Then, a CS member takes over as new coordinator and decides about the transaction.

Assume that a coordinator makes a decision and due to its exclusion from view v_i, a new coordinator takes over and makes a different decision. Let us assume without loss of generality that the new coordinator takes over in view v_{i+1} (in general, it will be in v_{i+f}). The old coordinator can have decided to:

1. *Commit*. The old coordinator can only decide commit if all votes have been uni-delivered, and they all were affirmative. If the new coordinator decides to abort, it can only be because it has not uni-delivered one or more votes neither before the view change nor before its timer expires. In this situation, there are two cases to consider:

   a. The new coordinator belonged to v_i. Hence, all votes were uni-delivered in v_i at the old coordinator (which needed all yes votes to decide to commit) but not at the new coordinator (otherwise it would also decide to commit) what violates multicast uniformity.
   b. The new coordinator joined the CS group after a recovery or a partition merge. From the recovery procedure, the new coordinator has gotten the most up-to-date state during the state transfer triggered by the view change. If it decides to abort, it is because a process in view v_i (the one which sent its tables in the state transfer) did not uni-deliver one or more votes. This again violates uniformity and it is therefore impossible.
2. *Abort due to a no vote*. If the old coordinator aborts the transaction, it does so as soon as the no vote is opt-delivered (CS.A). In order to decide to commit, the new coordinator needs to uni-deliver all votes and that all votes are yes. Since a participant votes only once, this situation cannot occur (for it to occur, a participant needs to say no to the old coordinator and yes to the new one).
3. *Abort due to a timeout*. If the old coordinator decided to abort due to a timeout, then it uni-delivered its own timeout message (CS.C). If the new coordinator decides to commit, then it must have uni-delivered all the votes before its timer expires and before uni-delivering the timeout message. This implies that the timeout message has been delivered to the old coordinator in view v_i and not to the new coordinator. Now there are two cases to consider:
   a. If the new coordinator was in view v_i, the fact that the old coordinator has not received the timeout message violates multicast uniformity. It is therefore not possible for the new coordinator to have been in v_i.
   b. If the new coordinator was not in view v_i then it has joined the group in view v_{i+1}, and thus during the state synchronization it has received the most up-to-date tables. However, this implies that some process in the CS group was in view v_i, transited to v_{i+1}, but did not uni-deliver the timeout message. Again this violates uniformity.

From here, since in all possible cases the new coordinator cannot make a different decision than the old coordinator once the latter has made a decision (abort or commit), the lemma is proven. □

Lemma 3 (NBAC-Termination). *If there is a time after, which there is a majority view sequence in the CS group that permanently contains at least a correct process, then the protocol terminates.* □

Proof (lemma 3): Assume for contradiction that the protocol never ends. This means that either:

– A correct coordinator never decides. A correct coordinator will either: (1) Opt-deliver a no vote, in which case the transaction is aborted, or (2) uni-deliver all the votes and are all yes, in which case the transaction is committed, or (3) uni-deliver the timeout message before opt-delivering all the votes (and being all previous votes affirmative), in which case the transaction is aborted. Therefore, if there is a correct process, there will eventually be a correct coordinator that will decide the transaction outcome and multicast it to the participants, thus terminating the protocol.

– There is an infinite sequence of unsuccessful coordinators that do not terminate the protocol. The NewCoordinator function, whenever possible, chooses a fresh coordinator (a process that did not previously coordinate the transaction). This means that a correct process p will eventually coordinate the transaction. Since p is correct and belongs to the majority view, it will eventually terminate the protocol as shown before.

Thereby, it is proven that the protocol eventually terminates. □

Lemma 4 (NBAC-Non-Triviality). *If all participants votes yes there are no failures nor false suspicions then commit will be decided.* □

Proof (lemma 4): The abort decision can only be taken when the coordinator receives a no vote, or because it times out. Otherwise, the decision is commit. □

Theorem 1 (NBAC-Correctness). *The protocol presented in the paper fulfills the non-blocking atomic commitment properties: NBAC-Validity, NBAC-Uniform-Agreement, NBAC-Termination, and NBAC-Non-Triviality.* □

Proof (theorem 1): It follows from lemmas 1, 2, 3, and 4 □

5 Conclusions

Atomic commitment is an important feature in distributed transactional systems. Many commercial products and research prototypes use it to guarantee transactional atomicity (and, with it, data consistency) across distributed applications. The current standard protocol for atomic commitment is 2PC which offers reasonable performance but might block when certain failures occur. In this paper we have proposed a non-blocking atomic commitment protocol that offers the same reasonable performance as 2PC but that is non-blocking. Unlike previous work in the area, we have emphasized several practical aspects of atomic commitment. First, the new protocol does not create any additional message overhead when compared with 2PC. Second, by using a replicated group as stable memory instead of having to flush log records to the disk, the protocol is likely to exhibit a shorter response time than standard 2PC. Third, the fact that the second round of the commit protocol is run only by a small subset of the participants minimizes the overall overhead. Fourth, and most relevant in practice, the new protocol can be implemented on top of the same interface as that used for 2PC. This is because, unlike most non-blocking protocols that have been previously proposed, the participants only need to understand a *prepare to commit* message (or *vote-request*) and then a *commit* or *abort* message. This is exactly the same interface required for 2PC and it is implemented in all transactional applications. Because of these features, we believe the protocol constitutes an important contribution to the design of distributed transactional systems. We are currently evaluating the protocol empirically to get performance measures and are looking into several possible implementations to further demonstrate the advantages it offers.

References

1. O. Babaoglu and S. Toueg. Understanding Non-Blocking Atomic Commitment. In *Distributed Systems*. Addison Wesley, 1993.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.
3. K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
4. K.P. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, NJ, 1996.
5. T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, March 1996.
6. C. Dwork and D. Skeen. The Inherent Cost of Nonblocking Commitment. In *Proc. of ACM PODC*, pages 1–11, 1983.
7. R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical report, CS Dep., Cornell Univ., 1995.
8. J. Gray. *Notes on Data Base Operating Systems. Operating Systems: An Advanced Course*. Springer, 1978.
9. R. Guerraoui. Revistiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. In *Proc. of the WDAG'95*, 1995.
10. R. Guerraoui, M. Larrea, and A. Schiper. Non-Blocking Atomic Commitment with an Unreliable Failure Detector. In *Proc. of the 14th IEEE Symp. on Reliable Distributed Systems*, Bad Neuenahr, Germany, September 1995.
11. R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting Commit Processing in Distributed Database Systems. In *Proc. of the ACM SIGMOD*, 1997.
12. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, pages 97–145. Addison Wesley, 1993.
13. I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit at No Additional Cost. In *Proc. of ACM PODS*, 1995.
14. I. Keidar and S. Rajsbaum. On the Cost of Faul-Tolerant Consensus Where There No Faults - A Tutorial. Technical Report MIT-LCS-TR-821, 2001.
15. B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431, 1999.
16. C. Mohan, B. Lindsay, and R. Obermark. Transaction Management in the R* Distributed Database System. *ACM Transactions on Database Systems*, 11(4), February 1986.
17. M. Patiño Martínez, R. Jiménez Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*, volume LNCS 1914, pages 315–329, October 2000.
18. F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In S. Kutten, editor, *Proc. of 12th Distributed Computing Conference*, volume LNCS 1499, pages 318–332. Springer, September 1998.
19. A. Schiper and A. Ricciardi. Virtually Synchronous Communication Based on a Weak failure Suspector. In *Proc. of FTCS-23*, pages 534–543, 1993.
20. A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proc. of ICDCS-13*, pages 561–568, 1993.
21. D. Skeen. Nonblocking Commit Protocols. *ACM SIGMOD*, 1981.
22. D. Skeen. A Quorum-Based Commit Protocol. In *Proc. of the Works. on Distributed Data Management and Computer Networks*, pages 69–80, 1982.
23. R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report CS99-31, Hebrew Univ., September 1999. To be published in ACM Comp. Surveys.

Un algoritmo $n\log(n)$ para la resolución del interbloqueo

A. Córdoba, F. Fariña, J.R. Garitagoitia, J.R. González de Mendívil, and J. Villadangos

Univ. Púb. de Navarra, Dpto. de Matemática e Informática, 31006 Pamplona.

Abstract. En este trabajo se presenta un nuevo algoritmo de resolución del interbloqueo para el modelo único-recurso basado en la historia de las sondas (*history-based algorithm*). El algoritmo tiene un peor caso de $\mathcal{O}(n \log n)$ en número de mensajes para un interbloqueo de n nodos. Para conseguir este número reducido de mensajes se propone, por una parte, disminuir el número de iniciaciones del algoritmo y, por otra, disminuir la distancia que puede recorrer un mensaje utilizando el concepto de 'nivel' alcanzado por cada nodo en el interbloqueo.

Palabras clave. Detección del interbloqueo, Resolución del interbloqueo, Modelo único recurso, Sistemas distribuidos, Algoritmos basados en la historia, Autómatas de E/S.

1 Introducción

En un sistema distribuido, cuando un proceso ejecutándose en un lugar del sistema necesita un recurso en otro lugar, envía un mensaje de petición de acceso al recurso y espera hasta que esa solicitud sea satisfecha. En este contexto, es posible que varios procesos queden en situación de interbloqueo si todos ellos esperan por recursos que mantienen otros procesos en el interbloqueo.

En el pasado se han desarrollado distintos algoritmos para detectar el interbloqueo en sistemas distribuidos [4][7]. Gran parte de ellos se basan en la utilización del denominado Grafo de Esperas (*Wait For Graph*- WFG), donde los nodos representan procesos y recursos, y los arcos las relaciones de espera entre estos [1][4]. El método más extendido entre los algoritmos de detección del interbloqueo en sistemas distribuidos es el denominado '*edge-chasing*' que se basa en el envío de pequeños mensajes denominados 'sondas' [2][3][5][8]. Un proceso bloqueado envía una sonda a través de los arcos, que es retransmitida a su vez por los procesos en espera. La situación de interbloqueo se detecta cuando la sonda llega al proceso que la inició.

Uno de los inconvenientes de este tipo de algoritmos es el número de mensajes necesario para la detección, pues, en general, es necesario recorrer todos los arcos del grafo, resultando un número de mensajes proporcional al de arcos en el grafo. Además el algoritmo puede ser iniciado por varios procesos a la vez, por lo que habría que multiplicar esta cifra por el número de iniciadores para obtener el total de mensajes para la detección. Normalmente el coste de estos algoritmos en

número de mensajes es $\mathcal{O}(n^2)$, siendo n el número de nodos en el interbloqueo. Una variante de estos algoritmos se presenta en [9]. En dicha variación se permite que un nodo, que no está directamente implicado en el interbloqueo, lo detecte y envíe el mensaje de resolución a algún nodo en el interbloqueo. La mejora propuesta reduce el coste desde un punto de vista estadístico, ya que depende de la probabilidad de que un nodo exterior inicie la fase final de detección del interbloqueo, pero el peor caso (usado habitualmente como medida de bondad) sigue siendo $\mathcal{O}(n^2)$. Otra forma de reducir el coste del caso promedio es emplear 'algoritmos basados en la historia' (*history-based algorithms*) cuyo objetivo es eliminar la necesidad de enviar la misma sonda más de una vez a través del mismo arco [3][5][8][9]. Para un interbloqueo de n nodos, el coste de estos algoritmos es $n \cdot (n-1)$ en el peor caso, y $2 \cdot (n-1)$ en el mejor.

En este artículo, presentamos un nuevo algoritmo de resolución del interbloqueo en sistemas distribuidos 'basado en la historia' para el modelo de petición único-recurso. Se hace especial énfasis en la descripción del algoritmo y en la evaluación de sus costes. El algoritmo necesita $n \cdot \log n$ (n número de procesos en el interbloqueo) mensajes para la detección en el peor caso, mejorando sensiblemente otras propuestas. El algoritmo es seguro (no resuelve falsos interbloqueos) y tiene viveza (resuelve todos los interbloqueos en tiempo finito), no obstante, no se incluye la demostración formal de estos aspectos en este trabajo.

El resto del artículo está organizado como sigue: La sección 2 introduce el modelo del sistema; En la sección 3 se describe el algoritmo de resolución del interbloqueo propuesto y algunas de las características de su funcionamiento; La sección 4 introduce el estudio del coste del algoritmo; Para acabar, la sección 5 contiene las conclusiones del trabajo.

2 Modelo del sistema

En esta sección presentamos los detalles del modelo desarrollado para el sistema de petición único-recurso al que va dirigido nuestro algoritmo de resolución de interbloqueo. También damos una caracterización de la situación de interbloqueo en dicho modelo único-recurso, así como los criterios de corrección que debe seguir todo algoritmo de resolución para el modelo planteado.

Un sistema distribuido puede verse como un conjunto de lugares en los que se ejecutan procesos que comparten un grupo de recursos. Sin pérdida de generalidad, suponemos que en cada lugar sólo existe un proceso o un recurso. Los procesos pueden acceder a los recursos de forma independiente a los lugares en los que residen. Suponemos que en cada lugar existe un gestor de recursos encargado del acceso al recurso del lugar. En el modelo de petición único-recurso, un proceso en una solicitud puede requerir el acceso a solamente un recurso. Un proceso que solicita un recurso queda bloqueado hasta que obtiene el acceso al recurso. Un recurso no puede estar asignado a más de un proceso a la vez. No vamos a diferenciar entre el recurso y el proceso gestor del recurso, por lo que hablaremos de esperas entre procesos, independientemente de su naturaleza y objetivos.

El AWFG es una pareja $(N \cup E, Arcs)$, $E = \{e_{ij} \mid i \neq j \wedge i, j \in N\}$, $Arcs \subseteq \{\langle i, e_{ij}\rangle, \langle e_{ij}, j\rangle \mid i \neq j \wedge i, j \in N\}$
N conjunto de *nodos* (procesos y recursos del sistema).
E conjunto de *nodos intermedios*.
Los arcos $\langle i, e_{ij}\rangle$, $\langle e_{ij}, j\rangle$ representan 'i espera a j'.
$E_i = \{e_{ij} \mid i \neq j \wedge j \in N\}$ conjunto de las esperas posibles del nodo i.
$Tarcs = \{\langle i, e_{ij}\rangle, \langle e_{ij}, j\rangle \mid i \neq j \wedge i, j \in N\}$ arcos posibles en el AWFG.

Fig. 1. Definición formal del Grafo de Esperas Asíncrono (AWFG).

La comunicación entre los distintos lugares del sistema se realiza a través de canales de comunicación. Se supone que existe un canal lógico de comunicación entre cualesquiera dos lugares del sistema y que estos son seguros. No hay memoria compartida entre los procesos, y la comunicación entre estos sólo es posible mediante el envío y recepción de mensajes.

En cada lugar del sistema existe una copia del algoritmo de resolución que proponemos en este trabajo. Para la resolución de un interbloqueo, el algoritmo aborta alguno de los procesos involucrados. Un proceso aborta liberando todos los recursos que mantiene ocupados y anulando la solicitud pendiente. Un proceso abortado 'desaparece' del sistema. Se supone que el sistema de nombrado de procesos (y prioridades) consigue que el algoritmo nunca aborte procesos gestores de recursos. Las únicas aborciones posibles en el sistema son las debidas al propio algoritmo de resolución (se supone que no hay aborciones espontáneas).

En la literatura sobre el tema, con el propósito de modelar la situación de interbloqueo se utiliza algun tipo de grafo. Generalmente se utiliza el denominado Grafo de Esperas (*Wait For Graph*- WFG) [1][4]. En el WFG un nodo representa un proceso (esto incluye a los gestores de recursos) y un arco $\langle i, j\rangle$ entre dos nodos i y j representa (i) una solicitud de un recurso del proceso i al proceso j (gestor del recurso) que no ha sido satisfecha; y (ii) una espera del proceso i (gestor de un recurso) al proceso j que ha obtenido el acceso al recurso y lo está ocupando (aún no lo ha liberado). En ambos casos, un arco en el WFG representa el 'reconocimiento' de una espera en una visión global del sistema. En trabajos anteriores, con objeto de representar el 'conocimiento local' de una espera, es decir la información que tiene la copia del algoritmo de resolución de un lugar, hemos propuesto utilizar el denominado Grafo de Esperas Asíncrono (*Asynchronous Wait For Graph*- AWFG) [10]. En el AWFG (figura 1), para cada pareja de nodos i y j que representan procesos, se introduce un nodo adicional e_{ij} denominado nodo intermedio que permite representar una espera entre el proceso i y el proceso j mediante dos arcos, el primero $\langle i, e_{ij}\rangle$, entre el nodo i y el nodo e_{ij}, representa el conocimiento que tiene el lugar en el que reside i de que este nodo espera por j, el segundo arco $\langle e_{ij}, j\rangle$, entre el nodo e_{ij} y el nodo j, representa el conocimiento que tiene el lugar en el que reside j de que está siendo esperado por el nodo i.

Un estado s del Autómata de E/S para el Administrador del Sistema está definido por los valores de las variables:

$s.Arcs, s.victims, \forall \langle n, n' \rangle \in Tarcs : s.version(\langle n, n' \rangle)$.

Estado inicial: $s_0 \in start(M)$:

$s_0.Arcs = \emptyset$

$s_0.victims = \emptyset$

$\forall \langle n, n' \rangle \in Tarcs : s_0.version(\langle n, n' \rangle) = 0$

Fig. 2. Estados, y sus valores iniciales, del Autómata de E/S que modela el Administrador del Sistema.

A continuación vamos a tratar de describir aquellos aspectos del funcionamiento del administrador del sistema relevantes para el problema de resolución del interbloqueo. La especificación formal de las variables de estado del administrador del sistema puede verse en la figura 2. El administrador M, para controlar el estado de las esperas entre procesos, debe mantener actualizado el AWFG que representa las esperas (conjunto *Arcs* en la figura 2). Además, va a mantener información sobre los procesos abortados (conjunto *victims*) y va a controlar la versión de los arcos en el AWFG (función *version*). La especificación formal de las operaciones del administrador del sistema M para el mantenimiento del estado puede verse en la figura 3. Para esta especificación hemos utilizado Autómatas de E/S [6].

Se supone que cuando un proceso i solicita un recurso j (gestionado por el proceso j), esta solicitud se tramita a través del administrador del sistema del lugar en el que se encuentra el proceso i que será el encargado de dirigirla al lugar del proceso j, por otra parte debe dejar constancia de que se ha iniciado una espera añadiendo el arco $\langle i, e_{ij} \rangle$ al AWFG (acción $StartAddArc_i(j)$ en la figura 3). Cuando el administrador del lugar del proceso j recibe dicha solicitud y esta no puede ser satisfecha dejará constancia de la espera en el AWFG añadiendo el arco $\langle e_{ij}, j \rangle$ (acción $EndAddArc_j(i)$). Cuando la solicitud pueda satisfacerse, el administrador del lugar j eliminará el arco $\langle e_{ij}, j \rangle$ del AWFG (acción $StartDelArc_j(i)$) y se pondrá en contacto con el del lugar i para que libere al proceso y consecuentemente elimine el arco $\langle i, e_{ij} \rangle$ del AWFG (acción $EndDelArc_i(j)$). Se supone que cada administrador local es el encargado de mantener la parte del AWFG que hace referencia a su lugar.

En estas condiciones, una *espera consistente* entre dos procesos en el sistema viene representada por dos arcos $\langle i, e_{ij} \rangle$ y $\langle e_{ij}, j \rangle$, con las condiciones de que tanto i como j no representen procesos abortados (i y j no deben pertenecer a *victims*) y que las versiones de ambos arcos coincidan, es decir, los arcos representan la misma solicitud no satisfecha. En nuestro grafo, la situación de interbloqueo está caracterizada por un *ciclo estable*, es decir un ciclo en el que todos sus arcos representan esperas consistentes. En ausencia de aborciones, la existencia de ciclos es una propiedad estable. La aborción de un proceso i es una operación de resolución del interbloqueo si y sólo si i pertenece a un ciclo estable.

Signatura:
$in(M) = \{\forall i \in N : Abort_i\}$
$out(M) = \{\forall i, j \in N, i \neq j :$
$StartAddArc_i(j), EndAddArc_i(j), StartDelArc_i(j), EndDelArc_i(j)\}$

Actions:
StartAddArc$_i(j)$
$pre \equiv d^+(i) = 0 \wedge i \notin victims.$
$eff \equiv Arcs \leftarrow Arcs \cup \{\langle i, e_{ij}\rangle\};$
$version(\langle i, e_{ij}\rangle) \leftarrow version(\langle i, e_{ij}\rangle) + 1.$

EndAddArc$_i(j)$
$pre \equiv j \notin victims \wedge version(\langle e_{ji}, i\rangle) = version(\langle j, e_{ji}\rangle) - 1.$
$eff \equiv Arcs \leftarrow Arcs \cup \{\langle e_{ji}, i\rangle\};$
$version(\langle e_{ji}, i\rangle) \leftarrow version(\langle e_{ji}, i\rangle) + 1.$

StartDelArc$_i(j)$
$pre \equiv d^+(i) = 0 \vee j \in victims.$
$eff \equiv Arcs \leftarrow Arcs - \{\langle e_{ji}, i\rangle\}.$

EndDelArc$_i(j)$
$pre \equiv d^+(e_{ij}) = 0 \wedge (version(\langle i, e_{ij}\rangle) = version(\langle e_{ij}, j\rangle) \vee j \in victims).$
$eff \equiv Arcs \leftarrow Arcs - \{(i, e_{ij})\}.$

Abort$_i$
$eff \equiv victims \leftarrow victims \cup \{i\};$
$Arcs \leftarrow Arcs - (incoming(i) \cup outgoing(i)).$

$d^+(i)$ representa el grado de salida del nodo i (número de arcos que salen del nodo)

Fig. 3. Acciones del Autómata de E/S que modela el Administrador del Sistema.

3 Algoritmo de resolución

El algoritmo propuesto es un algoritmo 'basado en la historia' [9], que utiliza una serie de mensajes denominados sondas que se transmiten entre los procesos en sentido contrario a los arcos del AWFG. El algoritmo interactúa con el administrador del sistema tal como se muestra en la figura 4. Cada proceso puede iniciar una instancia del algoritmo de resolución en un nivel dado (inicialmente el nivel cero), pero por efecto de la ejecución de otra instancia del algoritmo iniciada en otro proceso, puede subir de nivel e iniciar nuevas instancias. Así a cada ejecución de una instancia del algoritmo de resolución corresponde un proceso iniciador y un nivel del iniciador. La computación de una instancia del algoritmo de resolución se realiza entre todos los procesos que reciben las sondas generadas por el propio algoritmo.

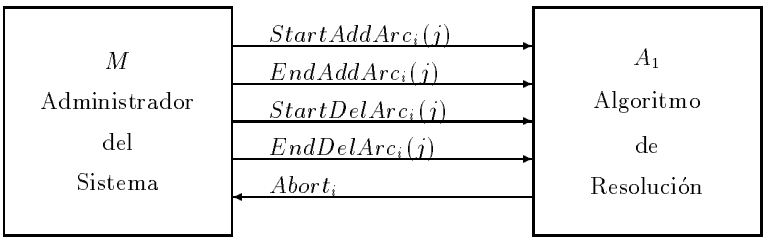


Fig. 4. Interface Administrador del Sistema/ Algoritmo de Resolución.

Un estado s del Autómata de E/S para el Algoritmo de Resolución está definido por los valores de las variables:
$s.Arcs, \forall i \in N : s.status_i, s.count_i, s.succ_i, s.remainning_i, s.smallers_i, \forall i, j \in N, i \neq j : s.channel_{ij}$.

$status_i$ indica el estado del nodo con respecto al algoritmo de resolución del interbloqueo.
$status_i \in$ {active, blocked, deadlock, level#x, level#x_received, victim}; x = 0 ...
$remaining_i$ conjunto de antecesores inmediatos del nodo i a los que i está pendiente de enviar una sonda.
$smallers_i$ antecesores inmediatos del nodo i de menor prioridad que este, a los que i está pendiente de enviar una sonda.
$count_i$ contador asociado al nodo i.
$succ_i.ident$ es el identificador de alguno de los sucesores del nodo i (o nil).
$succ_i.count$ es la cuenta correspondiente al sucesor en $succ_i.ident$.
$channel_{ij}$ es el canal desde el nodo i al nodo j.

Sonda $m \equiv (m.ident, m.type, m.count)$.
Tipos de sondas: *reference*, *level#x*, *recovery*; x = 0 ....

Fig. 5. Estados del Autómata de E/S del Algoritmo de Resolución.

El algoritmo utiliza en cada nodo las variables descritas en la figura 5. La variable $status_i$ indica el estado del nodo i con respecto al algoritmo de resolución, esta variable tiene uno de los valores siguientes: *active*, si el nodo no está en espera por ninguna solicitud, *level#x*, si el nodo ha iniciado una instancia del algoritmo de resolución en el nivel x, *blocked*, si el nodo espera por otro nodo, *deadlock*, si el nodo ha sido seleccionado para romper un ciclo y, finalmente, *victim*, si el nodo ha sido abortado. Una sonda m lleva el identificador ($m.ident$) y la cuenta ($m.count$) del nodo que ha iniciado la instancia del algoritmo y el tipo de sonda ($m.type$). La recepción de una sonda de tipo *reference* provoca, bajo ciertas condiciones, que el nodo que la recibe inicie una instancia del algoritmo de resolución. Un nodo con *status* en *level#x* envía sondas del tipo *level#x*. La funcionalidad de otras variables referidas en la figura 5 se verá más adelante a medida que vaya siendo necesario.

Signatura:
$in(A_1) = \{\forall i, j \in N, i \neq j :$
$\quad StartAddArc_i(j),\ EndAddArc_i(j),\ StartDelArc_i(j),\ EndDelArc_i(j)\}$
$out(A_1) = \{\forall i \in N :\ Abort_i\}$
$int(A_1) = \{\forall i, j \in N, i \neq j, \forall m \in S :\ ReceiveReference_i(j,m),\ SendLevelZero_i,$
$\quad SendLevel\#p_i,\ Detect_i(j,m),\ ReceiveLevel\#p_i(j,m),$
$\quad SendRemLevel\#p_i,\ ReceiveRecovery_i(j,m)\}$
Estado inicial s_0:
$s_0.Arcs = \emptyset$
$\forall i \in N :\quad s_0.status_i = active$
$\quad s_0.count_i = 0$
$\quad s_0.succ_i = (nil, 0)$ //es decir, $s_0.succ_i.ident = nil$ y $s_0.succ_i.count = 0$
$\quad s_0.remaining_i = s_0.smallers_i = \emptyset$
$\forall i, j \in N : s_0.channel_{ij} = \varepsilon$ //ε representa la cadena vacia

Fig. 6. Signatura y estados iniciales del Autómata de E/S del Algoritmo de Resolución.

La signatura del autómata de E/S que describe formalmente el algoritmo de resolución se muestra en la figura 6 y las acciones en las figuras 7 y 8. Algunas de las funciones utilizadas para la especificación de acciones son: *ant(i)* da el conjunto de los antecesores inmediatos del nodo i; *incoming(i)* da el conjunto de arcos que terminan en el nodo i; *outgoing(i)* da el conjunto de arcos que empiezan en el nodo i.

El algoritmo propuesto consta de dos fases. El objetivo de la primera fase es la detección de un posible interbloqueo. Esta fase termina con la clasificación de un único proceso como candidato a romper el interbloqueo detectado (*status = deadlock*). La segunda fase está dedicada a actualizar la información en los procesos de manera que, cuando el proceso aborte, el resto de los procesos del sistema queden en el estado que habrían alcanzado si el proceso abortado nunca hubiera existido. Es posible que varios procesos inicien instancias distintas del algoritmo de resolución, de tal forma que las fases de las distintas instancias del algoritmo se ejecutende forma paralela.

Cada espera entre dos procesos i y j se construye mediante dos acciones de forma asíncrona *StartAddArc$_i$(j)* y *EndAddArc$_j$(i)*, que representan el conocimiento de la existencia de esa espera por el proceso que espera y por el proceso esperado respectivamente. Se supone que la ejecución de la primera se origina por una solicitud de un recurso no satisfecha. La ejecución de la primera activa la ejecución de la segunda. La acción *EndAddArc$_j$(i)* envía hacia 'atrás' (hacia el proceso que queda en espera) una sonda tipo *reference* que, bajo determinadas condiciones, puede ser el detonante que pone en marcha una instancia del algoritmo de detección propiamente dicho. La recepción de la sonda *reference* (acción *ReceiveReference$_i$(j,m)*) permite definir la identidad del proceso sucesor ($succ_i.ident$) a la vez que se enumera ($succ_i.count$) la posible nueva instancia del algoritmo de detección a poner en marcha por el proceso.

StartAddArc$_i(j)$
eff $\equiv Arcs \leftarrow Arcs \cup \{\langle i, e_{ij}\rangle\}$;
  $status_i \leftarrow$ blocked.
EndAddArc$_i(j)$
eff $\equiv Arcs \leftarrow Arcs \cup \{\langle e_{ji}, i\rangle\}$;
  if $status_i \in \{$blocked, active$\}$ **then**
    $channel_{ij} \leftarrow channel_{ij} \bullet m$; // $m \equiv (i, \text{reference}, count_i)$
  $remaining_i \leftarrow remaining_i \cup \{j\}$;
  if $i > j$ **then** $smallers_i \leftarrow smallers_i \cup \{j\}$.
ReceiveReference$_i(j, m)$
pre $\equiv channel_{ji} = m \bullet \mu \wedge m.type =$ reference.
eff $\equiv channel_{ji} \leftarrow \mu$;
  $succ_i.ident \leftarrow m.ident$;
  $succ_i.count \leftarrow m.count$.
SendLevelZero$_i$
pre $\equiv i > succ_i.ident \wedge succ_i.ident \neq nil \wedge status_i \in \{$blocked, level#0$\} \wedge$
    $\wedge smallers_i \neq \emptyset$.
eff $\equiv$ **if** $status_i =$ blocked **then** $count_i \leftarrow count_i + 1$;
  $status_i \leftarrow$ level#0;
  $\forall k \in smallers_i$
    $channel_{ik} \leftarrow channel_{ik} \bullet m$; // $m \equiv (i, \text{level\#0}, count_i)$
  $remaining_i \leftarrow \emptyset$;
  $smallers_i \leftarrow \emptyset$.
SendLevel#p$_i$, p$\neq$0
pre $\equiv status_i =$ level#p $\wedge\ remaining_i \neq \emptyset$.
eff $\equiv \forall k \in remaining_i$
    $channel_{ik} \leftarrow channel_{ik} \bullet m$; // $m \equiv (i, \text{level\#p}, count_i)$
  $remaining_i \leftarrow \emptyset$;
  $smallers_i \leftarrow \emptyset$.
Detect$_i(j, m)$
pre $\equiv channel_{ji} = m \bullet \mu \wedge m.type \in \{$level#0 ... level#n$\} \wedge m.ident = i$.
eff $\equiv channel_{ji} \leftarrow \mu$;
  if $status_i \in \{$level#0 ... level#n$\} \wedge count_i = m.count$ **then**
    $status_i \leftarrow$ deadlock;
    $\forall k \in ant(i)$
      $channel_{ik} \leftarrow channel_{ik} \bullet m'$; // $m' \equiv (i, \text{recovery}, count_i)$
    $remaining_i \leftarrow \emptyset$;
    $smallers_i \leftarrow \emptyset$.
Abort$_i$
pre $\equiv \exists j \in N, m \in S : channel_{ji} = m \bullet \mu \wedge m.type =$ recovery $\wedge\ m.ident = i$.
eff $\equiv channel_{ji} \leftarrow \mu$;
  $status_i \leftarrow$ victim;
  $Arcs \leftarrow Arcs - (incoming(i) \cup outgoing(i))$;
  $succ_i.ident \leftarrow nil$;
  $succ_i.count \leftarrow 0$;
  $remaining_i \leftarrow \emptyset$;
  $smallers_i \leftarrow \emptyset$.

Fig. 7. Acciones del Autómata de E/S del Algoritmo de Resolución.

ReceiveLevel#p$_i(j,m)$
pre $\equiv channel_{ji} = m \bullet \mu \wedge m.type = \text{level\#p} \wedge m.ident \neq i.$
eff $\equiv channel_{ji} \leftarrow \mu;$
  if $status_i = \text{blocked} \vee (status_i = \text{level\#q} \wedge \text{level\#q} < \text{level\#p}) \vee$
    $(status_i = \text{level\#q\_received} \wedge \text{level\#q} < \text{level\#p})$ **then**
    $status_i \leftarrow \text{level\#p\_received};$
    $succ_i.ident \leftarrow m.ident;$
    $succ_i.count \leftarrow m.count;$
    $\forall k \in ant(i)$
      $channel_{ik} \leftarrow channel_{ik} \bullet m;$
    $remaining_i \leftarrow \emptyset;$
    $smallers_i \leftarrow \emptyset$
  else if $status_i = \text{level\#p} \wedge m.ident(-1)^{\#p} < i(-1)^{\#p}$ **then**
    $status_i \leftarrow \text{level\#p+1};$
    $remaining_i \leftarrow ant(i).$
SendRemLevel#p$_i$
pre $\equiv status_i = \text{level\#p\_received} \wedge remaining_i \neq \emptyset.$
eff $\equiv \forall k \in remaining_i$
    $channel_{ik} \leftarrow channel_{ik} \bullet m;$ // $m \equiv (succ_i.ident, \text{level\#p}, succ_i.count)$
  $remaining_i \leftarrow \emptyset;$
  $smallers_i \leftarrow \emptyset.$
ReceiveRecovery$_i(j,m)$
pre $\equiv channel_{ji} = m \bullet \mu \wedge m.type = \text{recovery} \wedge m.ident \neq i.$
eff $\equiv channel_{ji} \leftarrow \mu;$
  $succ_i.ident \leftarrow j;$
  $succ_i.count \leftarrow m.count;$
  if $status_i \in \{\text{level\#0} \ldots \text{level\#n}, \text{level\#0\_received} \ldots \text{level\#n\_received}\}$ **then**
    $status_i \leftarrow \text{blocked};$
    $\forall k \in (ant(i) - remaining_i)$
      $channel_{ik} \leftarrow channel_{ik} \bullet m';$ // $m' \equiv (m.ident, \text{recovery}, count_i)$
      if $k < i$ **then**
        $smallers_i \leftarrow smallers_i \cup \{k\};$
    $remaining_i \leftarrow ant(i).$
StartDelArc$_i(j)$
eff $\equiv Arcs \leftarrow Arcs - \{\langle e_{ji}, i\rangle\};$
  $remaining_i \leftarrow remaining_i - \{j\};$
  $smallers_i \leftarrow smallers_i - \{j\}.$
EndDelArc$_i(j)$
eff $\equiv status_i \leftarrow \text{active};$
  $Arcs \leftarrow Arcs - \{\langle i, e_{ij}\rangle\};$
  $succ_i.ident \leftarrow nil;$
  $succ_i.count \leftarrow 0;$
  $remaining_i \leftarrow \emptyset;$
  $smallers_i \leftarrow \emptyset.$

Fig. 8. Acciones del Autómata de E/S del Algoritmo de Resolución.

La decisión última para la puesta en marcha de una nueva instancia del algoritmo de detección por el proceso que ha recibido una sonda *reference* depende del cumplimiento de la precondición de la acción $SendLevelZero_i$. Si tenemos un camino de esperas $c \rightarrow b \rightarrow a$, en el que la última espera corresponde al arco $b \rightarrow a$, la acción $SendLevelZero_i$ está habilitada sólo si $b > c$ y $b > a$. Es evidente que en un camino cerrado (un ciclo como se ha visto equivale a una situación de interbloqueo) al menos existe un proceso b que cumple la condición de ser mayor que su predecesor y que su sucesor (en un ciclo de sólo dos procesos es evidente que la condición se cumple también). Es decir, si aparece un interbloqueo en el sistema al menos un proceso implicado inicia una instancia del algoritmo. Esta iniciación consiste en el envío de una sonda *level#0* a todos los procesos de identificador más pequeño que estén en espera por el proceso iniciador.

La recepción de una sonda *level#0* activa la acción $ReceiveLevel\#p_i(j,m)$ con $p = 0$. Supongamos que existe un camino $f \rightarrow e \rightarrow d \rightarrow c \rightarrow b \rightarrow a$, siendo b el proceso iniciador y que los procesos e y d tienen $status = bloqued$, entonces, primero el proceso c, seguido por el d y posteriormente el e, por medio de las correspondientes acciones $ReceiveLevel\#p_i(j,m)$ (con $p = 0$) reenvían la sonda (el c al d, el d al e y el e al f) además de guardar la identidad del proceso iniciador de la sonda ($succ_i.ident$) y el número de la instancia del algoritmo iniciado ($succ_i.count$). Si también existiera la espera $a \rightarrow f$, el proceso f reenviaría la sonda hacia el proceso a y este hacia el b. La recepción de la sonda en este último, activaría la acción $Detect_i$ haciendo que el proceso b detectase la existencia de un posible interbloqueo ($status = deadlock$). La ejecución de $Detect_i$ inicia la segunda fase mediante el envío de una sonda *recovery*.

Antes de continuar vamos a considerar algunos aspectos pasados por alto hasta ahora. Este es un algoritmo 'basado en la historia' que permite reducir el número total de sondas necesarias [9]. Si en el último ejemplo el proceso b (iniciador del algoritmo) recibiera nuevas esperas después de la ejecución de la acción $SendLevelZero_i$, estas volverían a activar esta acción (siempre bajo las mismas condiciones entre los identificadores de los procesos), de tal forma que la sonda *level#0* se transmitiría a todos los nuevos predecesores (menores) del proceso iniciador. Este mecanismo consigue que una sonda no se vuelva a enviar dos veces a través de un arco dado, a la vez que la sonda se envía a todos los predecesores independientemente del momento en que se produce la espera (siempre que el proceso tenga $status = level\#0$). Un mecanismo similar existe para aquellos procesos que han reenviado la sonda mediante la acción $ReceiveLevel\#p_i(j,m)$, la aparición de nuevas esperas a estos procesos activan en este caso la acción $SendRemLevel\#p_i$, con $p = 0$, haciendo que la sonda se transmita a todos los nuevos predecesores independientemente del momento en que se produce la espera (siempre que el proceso tenga $status = level\#0\_received$). En concreto, si en el ejemplo anterior el arco $a \rightarrow f$ apareciera después de que el proceso f reciba la sonda, activaría la acción $SendRemLevel\#p_i$ en el proceso f llegando a la misma situación anterior: la activación de la acción $Detect_i$ por el proceso b.

Otro aspecto a considerar es la posibilidad de que varios procesos pertenecientes a un interbloqueo activen concurrentemente la primera fase del algoritmo. Consideremos que existen las esperas $c \rightarrow b \rightarrow a$, y $f \rightarrow e \rightarrow d$, de tal manera que se cumple $b > c$, $b > a$ y $e > f$, $e > d$, los procesos b y e habrán iniciado sendas instancias del algoritmo con el envío de sondas *level#0* a los procesos c y f respectivamente. Estos procesos habrán pasado a *status = level#0_received* y habrán guardado el identificador del proceso iniciador, b y e respectivamente. Si posteriormente aparecen las esperas $d \rightarrow c$ y $a \rightarrow f$ (el interbloqueo existe), los procesos c y f reenvían las sondas de los respectivos iniciadores almacenados (acción $SendRemLevel\#p_i$, con $p = 0$). Finalmente, el proceso b recibirá la sonda iniciada por e, y el proceso e la sonda iniciada por b activándose la cláusula *else* de la acción $ReceiveLevel\#p_i(j,m)$, pero sólo uno, por ejemplo el b (supongamos $e < b$) pasaría a *status = level#1*, activándose en el proceso b la acción $SendLevel\#p_i$, con $p = 1$, y enviando una nueva sonda *level#1* a todos sus predecesores en el grafo de esperas. Esta sonda sería reenviada por todos los procesos en el interbloqueo, incluido el proceso e que se encontraría en *status = level#0* (todos los demás estarían en *status = level#0_received*) con lo que finalmente se activaría la acción $Detect_i$ para el proceso b.

Si existen m procesos de un interbloqueo que han iniciado su correspondiente instancia del algoritmo de detección, cuando cada uno de ellos recibe la sonda de otro (acción $ReceiveLevel\#p_i(j,m)$) tiene la posibilidad de permanecer en el nivel en el que se encuentra (cláusula *then* de la acción) y desactivar la instancia del algoritmo que él inició, o pasar al siguiente nivel (cláusula *else* de la acción) e iniciar una instancia del algoritmo al nuevo nivel. Un proceso que se encuentra a *status = level#p* no retransmite sondas de instancias del algoritmo iniciadas por otros procesos en un nivel inferior. De los m procesos que en un determinado momento se encuentran en un nivel dado p, a lo más $m-1$ procesos pueden pasar al nivel siguiente $p+1$, ya que la cláusula *else* de la acción $ReceiveLevel\#p_i(j,m)$ impone una relación de orden entre los m identificadores de los procesos (en realidad, como veremos a continuación, el proceso de eliminación es más rápido). Mediante este proceso de eliminación se consigue que sólo una instancia del algoritmo, la iniciada por el proceso que consigue un *status = level#p* mayor, termine la fase de detección y en consecuencia que sólo un proceso sea candidato para romper el interbloqueo.

Si n es el número de procesos implicados en un interbloqueo. El número de sondas *level#p* necesarias para llegar a activar la acción $Detect_i$ por un proceso implicado en el interbloqueo, en el caso de que sólo haya un proceso iniciador, es igual a n. Además, habría que considerar hasta n sondas *reference*, y cierto número de sondas *level#p* hacia procesos no implicados en el interbloqueo. En el ejemplo con dos procesos iniciadores propuesto, se han creado n sondas *level#0* y n sondas *level#1* (aparte de las enviadas a procesos no implicados), en total $2 \cdot n$ sondas.

En general, con n procesos $a_{n-1} \rightarrow a_{n-2} \rightarrow \ldots \rightarrow a_1 \rightarrow a_0 \rightarrow a_{n-1}$, implicados en un interbloqueo a lo más $n/2$ procesos cumplen las condiciones de la acción $SendLevelZero_i$ para activar el algoritmo ($a_i > a_{i+1 \bmod n}$, $a_i > a_{i-1 \bmod n}$).

Las sondas *level#0* generadas se reenvían hasta el siguiente proceso iniciador (en total se habrían creado $2 \cdot n/2$ sondas *level#0*). Los procesos iniciadores cuyo identificador es mayor que el de la sonda recibida pasarían al siguiente nivel *status = level#1* activando la acción $SendLevel\#p_i$, con $p = 1$, enviando una sonda *level#1*. ¿Cuantos procesos (de los $n/2$ iniciadores) pasarían a *status = level#1*?.

Consideremos dos situaciones límite. Supongamos n par y que los procesos iniciadores son aquellos de subindices pares, además $a_{n-2} > a_{n-4} > \ldots > a_2 > a_0$. En esta situación todos, menos el proceso a_0, reciben sondas *level#0* cuyo identificador es menor al del propio proceso y por lo tanto todos (menos el a_0) pasan a *status = level#1*. Esta situación se repite con las sondas *level#1*, todos los procesos menos el a_2 reciben sondas *level#1* cuyo identificador es menor al del propio proceso y por lo tanto (el signo de la comparación cambia en $ReceiveLevel\#p_i(j,m)$, con $p = 1$) este es el único proceso que pasa a *status = level#2*. Ahora todos los procesos salvo a_2, están en un *status = level#p* con $p < 2$, y se limitan a reenviar la sonda *level#2* enviada por este. En total se han creado $3 \cdot n$ sondas (n sondas de cada nivel hasta *level#2*) para conseguir activar la acción $Detect_i$ en algún proceso. Considerando la misma situación pero ahora $a_0 > a_2 > \ldots > a_{n-4} > a_{n-2}$. Ahora todos, menos el proceso a_0, reciben sondas *level#0* cuyo identificador es mayor al del propio proceso y por lo tanto el único proceso que pasa a *status = level#1* es el a_0, siendo este el único que activa $SendLevel\#p_i$, con $p = 1$. Ahora sólo son necesarias $2 \cdot n$ sondas en total para hacer que el proceso a_0 active la acción $Detect_i$.

En general, si un proceso a con un determinado *status = level#p* con p par (impar), recibe una sonda desde un proceso b, cuyo identificador es menor (mayor) que el identificador del proceso, lo que le permite pasar a *status = level#p+1*, $p + 1$ es impar (par), en el nuevo nivel una sonda originada en b no provoca que a pase a *status = level#p+2*, ya que la relación ahora debería ser b mayor (menor) que a. La única posibilidad de que el proceso a pase al nivel *status = level#p+2*, es que la sonda llegue a través de b desde otro proceso c que también se encuentre en *status = level#p+1*, y que cumpla la condición c mayor (menor) que a. Esto sólo es posible si el proceso b permaneció en *status = level#p*, es decir desistió en la búsqueda del interbloqueo. El caso peor corresponde a la situación en la que en cada nivel se eliminan a lo más la mitad de los procesos de ese nivel, lo que hace necesarias $n \cdot \log n$ sondas para conseguir que un proceso active la acción $Detect_i$.

Una vez se activa la acción $Detect_i$ en algún proceso, este detecta el interbloqueo (*status = deadlock*) y toma la responsabilidad, primero de devolver al resto de los procesos al estado previo a la ejecución del algoritmo y, segundo de romper el interbloqueo. Con el primer objetivo envía una sonda *recovery* a todos los procesos que lo esperan. La recepción de una sonda *recovery* activa la acción $ReceiveRecovery_i(j,m)$ cuyo fin es restaurar el *status = blocked* y los distintos conjuntos de predecesores utilizados en el algoritmo, además reenvía la sonda *recovery* a todos sus predecesores. Cuando una sonda *recovery* se recibe

por el proceso que la inició (la sonda ha dado la vuelta al ciclo) se activa la acción $Abort_i(j,m)$ que rompe el ciclo.

La espera entre procesos se destruye mediante dos acciones de forma asíncrona $StartDelArc_i(j)$ y $EndDelArc_i(j)$, la primera se realiza por el proceso esperado (el primero que 'conoce' que la espera ha terminado) y la segunda por el proceso que espera (finalizando así la espera). La ejecución de la primera origina la ejecución de la segunda.

4 Medidas de complejidad

Para resolver el problema de la complejidad del algoritmo, vamos a tratar de contestar a la siguiente pregunta: ¿Cuál es el número mínimo de nodos involucrados en un interbloqueo necesarios para que otro nodo i, en el mismo interbloqueo, verifique $status(i) = level\#n$?.

Sea un circuito de interbloqueo en el que existe un nodo i con $status(i) = level\#n$. Supongamos n impar, si existe un nodo en este estado es porque al menos existen dos nodos i_1 y i_2 de tal manera que siendo $id(i_1) > id(i_2)$ (es decir, i_1 recibe una sonda i_2 menor que él) y $i = i_1$ verifican $status(i_1) = status(i_2) = level\#n-1$. Para que estos dos nodos hayan alcanzado ese estado, es necesario que existan al menos tres nodos i_3, i_4 y i_5, de tal manera que $id(i_3) < id(i_4)$, $id(i_4) > id(i_5)$, (es decir, i_3 recibe una sonda i_4 mayor que él, y i_5 recibe una sonda i_3 mayor que él) con $i_3 = i$ y $i_5 = i_2$, y verifiquen $status(i_3) = status(i_4) = status(i_5) = level\#n-2$. De idéntica forma, para que estos nodos estén en ese estado, deben existir al menos 5 nodos que verifican condiciones semejantes a las anteriores. Es decir, existen i_6, i_7, i_8, i_9 y i_{10}, de tal manera que $id(i_6) > id(i_7)$, $id(i_7) > id(i_8)$, $id(i_8) < id(i_9)$, $id(i_9) > id(i_{10})$ y $id(i_{10}) < id(i_6)$ con $i_6 = i_3$, $i_7 = i_5$ y $i_9 = i_4$. La relación de las sondas es: i_6 recibe una sonda menor que él (i_{10}), i_7 recibe una sonda mayor que él (i_6), i_8 recibe una sonda mayor que él (i_7), i_9 recibe una sonda menor que él (i_8) y i_{10} recibe una sonda mayor que él (i_9). Todos ellos verifican que su estado es $level\#n-3$. Así se procederá de forma sucesiva.

En definitiva para obtener un nodo en el estado $level\#n$, se necesitan un número mínimo de nodos que verifican la sucesión: 1, 2, 3, 5, 8, 13,..... Esta sucesión coincide con la sucesión de *Fibonacci*. Es decir, si existe un nodo i en el circuito de interbloqueo con $status(i) = level\#x$ entonces para todo entero y, $y \leq x$, existieron un conjunto de nodos N_y tales que alcanzaron $level\#(x-y)$, siendo $cardinal(N_y) \geq fib(y+2)$. En particular, existió un conjunto de nodos N_0 tales que alcanzaron $level\#0$, siendo $cardinal(N_0) \geq fib(x+2)$. Por lo tanto, si tenemos n nodos en el $level\#0$, podemos asegurar que a lo más algún proceso puede alcanzar un estado $level\#l$ tal que $n \geq fib(l+2)$.

Sea un ciclo con n nodos, en el peor caso pueden llegar $n-1$ al estado $level\#0$. Por lo tanto, podemos acotar el mayor $level\#L$ por el valor $n-1 \geq fib(L+2)$. Por la expresión de *Fibonacci*, $fib(n) = c*(p^n - q^n)$ con $p = \frac{1}{2}+\frac{1}{2}\sqrt{5}$, $q = \frac{1}{2}-\frac{1}{2}\sqrt{5}$ y $c = \frac{1}{\sqrt{5}}$, podemos asegurar que $fib(L+2) \geq C^{L+2}$, siendo C una constante, por lo que $\log(n-1) \geq (L+2)\log C$ y en consecuencia $L \leq \frac{\log(n-1)}{\log C} - \log C^2 =$

$\frac{\log(\frac{n-1}{C^2})}{\log C}$. En conclusión, el máximo nivel que puede alcanzar un nodo es $\mathcal{O}(\log n)$ y, tal como se vio anteriormente, en cada nivel son necesarios n mensajes, con lo que el número de mensajes es $\mathcal{O}(n \cdot \log n)$.

5 Conclusiones

En este trabajo se ha presentado un nuevo algoritmo de resolución del interbloqueo en sistemas distribuidos 'basado en la historia' para el modelo de petición único-recurso. Este algoritmo presenta un coste en el número de mensajes $\mathcal{O}(n \cdot \log n)$ (peor caso) que rebaja sensiblemente el coste de otros algoritmos con el mismo fin propuestos por otros autores. Como trabajo futuro, queda pendiente desarrollar una prueba formal de la corrección del algoritmo.

6 Agradecimientos

Este trabajo ha sido parcialmente soportado por la CICYT, mediante el proyecto TIC-1999-0280-C02-02, y la Unión Europea, mediante el proyecto GLOBDATA número IST-1999-20997.

References

1. R.C. Holt; *Some Deadlock Properties on Computer Systems*; ACM Computing Surveys, vol. 4, n. 3, pp.179-196, September 1972
2. K.M. Chandy, J. Misra, L.M. Haas; *Distributed Deadlock Detection*; ACM Trans. on Computer Syst., vol. 1, no. 2, pp. 144-156. May 1983.
3. M.K. Sinha, N. Natarajan; *A Priority Based Deadlock Detection Algorithm*; IEEE Trans. on Softw. Eng., vol. 11, no. 1, pp. 67-80, Jan. 1985.
4. E. Knapp; *Deadlock Detection in Distributed Databases*; ACM Computing Surveys, vol. 19, no. 4, pp. 303-328, Dec. 1987.
5. M. Roesler, W.A. Burkhard; *Resolution of Deadlocks in Object-oriented Distributed Systems*; IEEE Trans. on Computers, vol. 38, pp. 1212-1224, Aug. 1989.
6. N.A. Lynch, M.R. Tuttle; *An Introduction to Input/Output Automata*; CWI Quarterly, vol. 2, no. 3, pp. 219-246, Sep. 1989.
7. M. Singhal; *Deadlock Detection in Distributed Systems*; IEEE Computer, vol. 22, pp. 37-48, Nov. 1989.
8. A.D. Kshemkalyani, M. Singhal; *Invariant-Based Verification of a Distributed Deadlock Detection Algorithm*; IEEE Trans. on Softw. Eng., vol. 17, no. 8, pp. 789-799, Aug. 1991.
9. S. Lee, J.L. Kim; *An Efficient Distributed Deadlock Detection Algorithm*; Proc. of the 15th Int. Conf. on Distributed Computing Systems, Vancouver, Canada, pp. 169-178, May 1995.
10. J.R. González de Mendívil, F. Fariña, J.R. Garitagoitia, C. F. Alastruey, J.M. Bernabeu-Auban; *A Distributed Deadlock Resolution Algorithm for the AND Model*; IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 5, pp. 433-447, May 1999.

Un entorno de programación distribuida adaptable a diferentes plataformas*

E. Martel, F. Guerra, and J. Miranda

Instituto Universitario de Microelectrónica Aplicada
Departamento de Ingeniería Telemática
Universidad de Las Palmas de Gran Canaria
{emartel,fguerra,jmiranda}@iuma.ulpgc.es

Abstract. En este artículo presentamos un entorno gráfico para programar aplicaciones distribuidas, el EPDA (*Environment for Programming Distributed Applications*). La principal característica de este entorno es que se puede personalizar a diferentes plataformas de programación distribuida mediante una herramienta de modelado. El entorno soporta el desarrollo y reconfiguración dinámica de aplicaciones distribuidas y permite una instalación tolerante a fallos para aquellos casos en los que se desee garantizar su continuidad en presencia de fallos en los nodos. La herramienta de modelado o *EPDA-Modeller* permite adaptar el EPDA a las características específicas de diferentes entornos de programación distribuida: Glade, la biblioteca de comunicaciones Group_IO, etc.

Keywords: configuración distribuida, Ada, Glade, Group_IO

1 Antecedentes

El enfoque de la programación de la configuración (*configuration programming* [1]) facilita el desarrollo y evolución de aplicaciones distribuidas. Este enfoque separa la estructura de una aplicación distribuida del comportamiento de sus componentes individuales, es decir, separa la programación de sus componentes de su configuración. La *programación de una aplicación distribuida* se consigue mediante un lenguaje de programación y los mecanismos de comunicación adecuados. La *configuración de una aplicación distribuida* describe la estructura de la aplicación en función de sus componentes software, sus interconexiones y la ubicación de esos componentes en los nodos del sistema. Esta configuración se puede realizar a mano ([2], [3], [4]), que resulta tedioso y propenso a errores. Por ello, muchos autores ([5], [6], [7]) optan por la utilización de lenguajes de configuración y entornos de programación asociados para facilitar la construcción, configuración y gestión de las aplicaciones distribuidas. Muchos entornos pueden tener una apariencia gráfica [8] y en algunos casos soportan la configuración de forma gráfica, sin necesidad de un lenguaje de configuración [9].

* *Este trabajo ha sido parcialmente financiado por el proyecto TIC2001–1586–C03–03 de la CICYT.*

La programación de aplicaciones distribuidas debe soportar cambios en su estructura durante su ejecución (*evolutionary changes* [10]). Estos cambios permiten recuperar el estado de la aplicación ante fallos del sistema, mejorar su disponibilidad, iniciar su migración e incluir nuevas funciones o tecnología. Todo ello sin interrumpir ni alterar aquellos componentes de la aplicación no implicados en el cambio. En cualquiera de los casos, la aplicación distribuida debe estar programada para soportarlos (por ejemplo, si se necesita incorporar una réplica a un grupo en ejecución, la nueva réplica puede necesitar el estado del grupo y el grupo, a su vez, debe tener la funcionalidad necesaria para ello). Los cambios en la estructura de una aplicación durante su ejecución los soportan los lenguajes de configuración de Durra [5], Darwin [6] y Glade [11], el entorno LuaSpace [12] y la plataforma ZCL [13]. Pirahna [14], herramienta diseñada para proporcionar disponibilidad a las aplicaciones CORBA, soporta los siguientes cambios durante la ejecución de las aplicaciones distribuidas: re-arranque de aquellos objetos que han fallado, actualización de la funcionalidad de sus componentes y aumento en la disponibilidad de la aplicación. El Manual de Referencia de Ada [15] no proporciona funciones específicas que permitan alterar la estructura de una aplicación durante su ejecución. Sin embargo, el programador de Ada puede utilizar los paquetes Shared_Passive, que se proporcionan en el Anexo para sistemas distribuidos del Manual de Referencia de Ada, para guardar el estado del servidor en ejecución. De esta forma si el servidor deja de funcionar se puede recuperar su estado y arrancarlo en otro lugar del sistema.

Este artículo presenta un entorno gráfico para programar aplicaciones distribuidas o EPDA que facilita la construcción, configuración, ejecución, monitorización y evolución de las aplicaciones distribuidas. El EPDA soporta el desarrollo y reconfiguración dinámica de aplicaciones distribuidas mediante un modelo cliente/servidor. Nuestro EPDA es tolerante a fallos, que resulta imprescindible para monitorizar y adaptar aplicaciones distribuidas tolerantes a fallos. En este artículo se presenta además una herramienta que personaliza el EPDA a una plataforma de programación distribuida concreta. Este herramienta de modelado de EPDAs la llamaremos *EPDA-Modeller*.

La estructura de este artículo se detalla a continuación. La sección 2 presenta cómo especificar la estructura de una aplicación distribuida. En la sección 3 se muestra la funcionalidad del EPDA. La sección 4 indica cómo se proporciona la funcionalidad del EPDA a través de su interfaz gráfica de usuario (GUI, *Graphical User Interface*). En la sección 5 proponemos una arquitectura cliente/servidor para el EPDA. La sección 6 presenta la herramienta de modelado del EPDA. La sección 7 muestra un ejemplo de uso de la herramienta de modelado y del EPDA construido a medida. Finalmente, se presentan las conclusiones y el trabajo actual.

2 Estructura de una aplicación distribuida

La estructura de una aplicación distribuida se puede especificar mediante la descripción de sus componentes y las interconexiones entre ellos. La descripción de

un componente se realiza en función de sus propiedades. Se puede considerar una aplicación distribuida como una jerarquía de componentes simples y complejos. Los *componentes simples* constituyen la unidad básica de distribución y aparecen en el nivel más bajo en la jerarquía de componentes. Los *componentes complejos* son una colección de componentes simples y complejos que se sitúan en aquellos niveles de la jerarquía en los que no aparecen los componentes simples. A continuación analizamos la estructura de aplicaciones distribuidas construidas mediante dos paradigmas de programación distribuida ampliamente conocidos: los grupos y la RPC. Las aplicaciones distribuidas basadas en grupos se han construido mediante la biblioteca de comunicaciones Group_IO [4], mientras que las aplicaciones distribuidas basadas en RPC se han construido en Glade ([7], [11]).

Los principales componentes de una *aplicación distribuida construida con Group_IO* son las *aplicaciones* y los *agentes*. Cada aplicación distribuida se compone de al menos un agente, que constituye la unidad básica de distribución y ejecución en Group_IO. Una colección de agentes se puede construir como un *grupo* [16]. Si una aplicación posee grupos, cada grupo debe contener como mínimo un agente.

Cada componente de una aplicación distribuida en Group_IO se describe mediante un conjunto de propiedades. Algunas de estas propiedades son las siguientes: nombre de la aplicación, localización de los ejecutables del agente, localización de arranque del agente, una marca para aquellos agentes catalogados como críticos en la aplicación (aquellos agentes que debe rearrancarse si fallan), el mínimo número de réplicas que necesita un grupo replicado para proporcionar su servicio, etc.

Las reglas de relación que deben seguir los componentes de las aplicaciones distribuidas construidas con Group_IO son las siguientes: una aplicación debe contener como mínimo un agente y opcionalmente una colección de agentes de la aplicación pueden formar un grupo; si existen grupos, cada grupo debe contener como mínimo un agente. Además, un agente puede formar parte de varios grupos. La notación UML que representa las reglas de relación entre los componentes de una aplicación distribuida en Group_IO se presenta en la figura 1, que también muestra algunas de las propiedades de sus componentes.

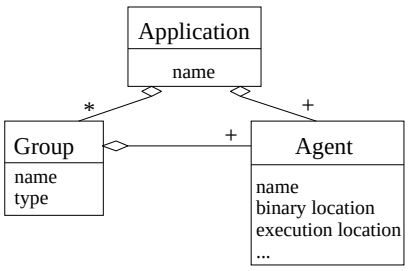


Fig. 1. Notación UML para Group_IO

Los principales componentes de una *aplicación distribuida construida en Glade* son los siguientes: *aplicaciones*, *particiones* y *unidades de biblioteca Ada*. Una aplicación distribuida en Glade está formada por particiones, que constituye la unidad básica de distribución en Glade. Una partición es una colección de unidades de biblioteca Ada. La conexión bi-direccional entre dos particiones forma un *canal*. Por tanto, las aplicaciones en Glade están formadas por particiones, unidades de biblioteca Ada y opcionalmente canales.

Las propiedades de una aplicación en Glade incluyen su nombre, el método de arranque, el nombre del procedimiento principal, la localización del *boot server*, los parámetros del *pool* de tareas, la localización de los ejecutables y dónde se deben arrancar, el control de versiones y el filtro de registro. Las propiedades de las particiones incluyen su nombre, el nombre de su procedimiento principal, la localización de los ejecutables y dónde se deben arrancar, las políticas de terminación y reconexión y el nombre de un filtro (si se necesita). Las propiedades de los canales incluyen el nombre de sus particiones y en caso de necesitarlo, el nombre del filtro. La notación UML que representa las reglas de relación entre los componentes de una aplicación distribuida en Glade se presenta en la figura 2, que también muestra algunas de las propiedades de sus componentes.

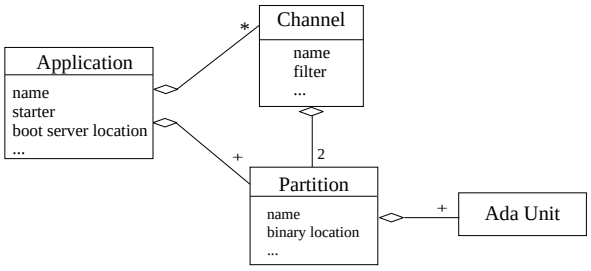


Fig. 2. Notación UML para Glade

3 Funcionalidad del EPDA

La funcionalidad del EPDA se consigue mediante las siguientes operaciones de gestión: construcción, configuración, ejecución, monitorización y evolución. Estas operaciones trabajan sobre los componentes que integran la aplicación distribuida. La *construcción* de aplicaciones distribuidas incluye el desarrollo de las entidades software, así como el establecimiento de los parámetros necesarios para generar los ejecutables de la aplicación. Las operaciones de construcción permiten programar los componentes de la aplicación distribuida mediante un lenguaje de programación y los mecanismos de comunicación adecuados. Las herramientas de programación necesarias (editor, compilador, depurador, etc.) pueden utilizarse desde el EPDA o mediante herramientas externas que dejan la

aplicación distribuida lista para que en el EPDA se realicen el resto de las tareas de gestión: configuración, ejecución, monitorización y evolución. La *configuración* de las aplicaciones distribuidas abarca la descripción de los componentes software mediante sus propiedades, la definición de las interconexiones entre los componentes y el uso de los recursos del sistema de acuerdo con los requisitos de la aplicación (por ejemplo, la localización de los componentes software en los nodos del sistema distribuido). Las operaciones de configuración incluyen añadir un nuevo componente a la aplicación, borrar, actualizar o revisar un componente que ya existe o bien comprobar si un nodo tiene los recursos necesarios para arrancar un componente. Todas estas operaciones tienen lugar antes de la ejecución de la aplicación. Las operaciones de ejecución de una aplicación distribuida incluyen el arranque y finalización de sus componentes. La *monitorización* de una aplicación distribuida permite consultar las propiedades de sus componentes antes de la ejecución (monitorización estática), así como el estado de la aplicación durante su ejecución (monitorización dinámica). Algunas situaciones que se detectan en la monitorización dinámica son las siguientes: los componentes que deben rearrancarse, el número de miembros que deben añadirse a un grupo para aumentar su disponibilidad, etc. La *evolución* de las aplicaciones distribuidas se refiere a los cambios que se realizan en la estructura de la aplicación durante su ejecución. Estas operaciones incluyen rearrancar un componente que ha fallado, incrementar la disponibilidad de un servicio, migrar componentes, etc.

Para facilitar las operaciones de configuración y evolución introducimos el concepto de *nodo lógico*. La unidad básica de distribución en una aplicación distribuida se ejecuta en un nodo lógico. Cada nodo lógico se asigna a un nodo físico del sistema distribuido. La distinción entre nodo físico y lógico nos permite tener dos vistas de un sistema distribuido: la vista física y la lógica. Independientemente de nuestro sistema distribuido real, la vista lógica nos permite definir los nodos lógicos en los que se ejecutará la aplicación, mientras que los nodos físicos del sistema distribuido junto con los nodos lógicos asociados se especifican en la vista física. Esta separación entre la vista física y lógica facilita la ubicación de las aplicaciones distribuidas en diferentes configuraciones del sistema distribuido. Por tanto, se ha conseguido un cierto grado de independencia entre las aplicaciones distribuidas y el hardware en el que se ejecutan. Así, por ejemplo, el programador puede ejecutar una aplicación distribuida de forma centralizada en una máquina con un único procesador. Para ello, todos los nodos lógicos utilizados por la aplicación distribuida se asocian al mismo nodo físico en tiempo de configuración. De esta forma se simplifica la depuración de las aplicaciones distribuidas. Cuando la ejecución centralizada de la aplicación distribuida funcione, el cambio a una ejecución distribuida es inmediato: basta con alterar la asociación entre los nodos lógicos y físicos y arrancar de nuevo la aplicación.

Nuestro EPDA pasa por los siguientes estados según actúen las operaciones de gestión sobre las aplicaciones distribuidas: *idle, building* y *running* (ver figura 3). Estos estados los utiliza el EPDA para diferenciar las operaciones permitidas. Así, las operaciones de construcción y configuración se permiten en los estados

idle y *building*. Las aplicaciones distribuidas se encuentran en el estado *building* cuando se están llevando a cabo las operaciones de construcción o configuración. Las operaciones de ejecución se permiten tanto en los estados *idle* como *running*. Las aplicaciones distribuidas pasan al estado *running* después de que la operación de arranque tenga lugar y al estado *idle* después de finalizar un componente en ejecución. Las operaciones de monitorización se pueden llevar a cabo en cualquier estado. Las operaciones de evolución sólo se permiten cuando las aplicaciones distribuidas se encuentran en el estado *running*.

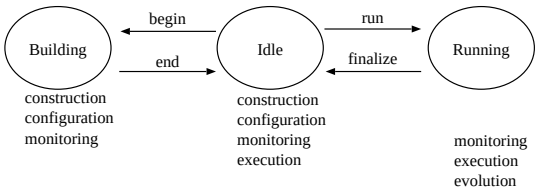
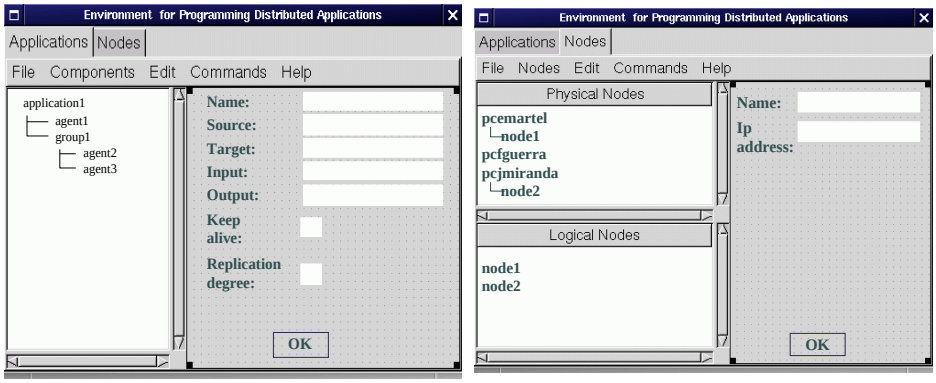


Fig. 3. Estados del EPDA

4 La GUI del EPDA

El EPDA proporciona su funcionalidad a través de una GUI, que se ha estructurado en dos secciones: la sección de aplicaciones y la sección de nodos. Cada sección tiene una barra de menú con las operaciones necesarias para la gestión de nodos y aplicaciones. La *sección de aplicaciones* consta de dos partes: la parte izquierda se utiliza para describir la estructura de las aplicaciones distribuidas; la parte derecha se utiliza para editar las propiedades de los componentes. Para describir la estructura de las aplicaciones distribuidas la notación UML se ha sustituido por una estructura en árbol por considerarse más amigable. El botón de confirmación de propiedades permite aceptar la descripción de la aplicación (sus componentes, propiedades y relaciones entre ellos). La GUI evita que el usuario viole las reglas de relación entre los componentes de una aplicación distribuida descrita en la sección 2. La figura 4-a muestra la GUI de la sección de aplicaciones para aplicaciones distribuidas construidas con Group_IO. La *sección de nodos* permite definir los nodos físicos y lógicos y la asociación entre ellos. La parte izquierda de la sección de nodos se utiliza para indicar los nombres de los nodos y la asociación entre los nodos lógicos y físicos, mientras que la parte derecha se utiliza para editar las propiedades de los nodos. El botón de confirmación de propiedades permite aceptar la descripción del nodo (sus propiedades y asociación entre los nodos lógicos y físicos). La GUI evita que el usuario asigne un nodo lógico a más de un nodo físico. En la figura 4-b se presenta la sección de nodos de la GUI para aplicaciones distribuidas construidas con Group_IO.



(a) Application section (b) Node section

Fig. 4. La GUI del EPDA para Group_IO

5 Arquitectura del EPDA

La arquitectura del EPDA se basa en el modelo cliente/servidor (ver figura 5). El cliente mediante la GUI envía peticiones al servidor, que es responsable de la construcción, configuración, ejecución, monitorización y evolución de aplicaciones distribuidas. El cliente proporciona la estructura de la aplicación al servidor, que la almacena en su base de datos. Se ha seleccionado una base de datos para mantener la información de cada componente en el disco por varias razones: proporciona una forma sencilla de consultar y actualizar la información de configuración; además, si el diseño de la base de datos es el adecuado, se evitan duplicidades y redundancia de los datos, reduciendo así el espacio necesario en disco.

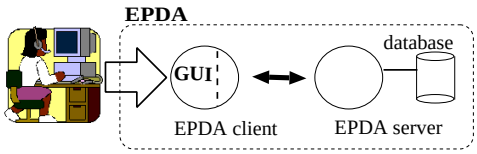


Fig. 5. Arquitectura cliente/servidor del EPDA

El usuario ejecuta las operaciones proporcionadas por la GUI del cliente sobre los componentes de la aplicación distribuida. La parte servidor del EPDA ejecuta las operaciones que le indica el cliente de acuerdo con el estado en que se encuentren las aplicaciones. Como el servidor puede recibir peticiones de diferentes usuarios desde diferentes partes del sistema distribuido, se necesita algún mecanismo de acceso concurrente a los datos. Para ello se utiliza el enfoque de los lectores/escritores: cualquier operación sobre un componente que implique

un cambio en la estructura de la aplicación bloquea cualquier otra operación sobre el mismo componente; sin embargo, las operaciones de consulta pueden operar concurrentemente.

5.1 EPDA cliente

El EPDA cliente se compone de dos partes: la GUI y los protocolos que permiten la comunicación con el EPDA servidor.

Cualquier descripción de un componente de la aplicación o de un nodo que se acepte desde la GUI se envía al EPDA servidor, que lo almacena en su base de datos. El resto de las operaciones proporcionadas por el EPDA sobre un componente seleccionado también se envían al servidor que retorna el estado de la operación cuando haya finalizado. Por ejemplo, para editar un componente (monitorización estática), el usuario lo selecciona y comunica la operación al servidor, que proporciona las propiedades del componente. El EPDA cliente muestra esas propiedades en la parte derecha de la sección de aplicaciones. Si el usuario las actualiza, los cambios se envían al servidor, que actualiza su base de datos. La comunicación entre el EPDA cliente y el servidor se realiza mediante la biblioteca de comunicaciones Group_IO [17].

5.2 EPDA servidor

El servidor mantiene el estado y la estructura de las aplicaciones distribuidas en su base de datos. Con el fin de evitar que el servidor sea un único punto de fallo, el servidor se implementa como un grupo replicado. Cada réplica se implementa mediante un código determinista que mantiene una copia local de la base de datos [18]. Para asegurar que todas las réplicas reciben exactamente la misma secuencia de peticiones en el mismo orden y de forma uniforme se utiliza la biblioteca de comunicaciones Group_IO. Además, de acuerdo al modelo de máquina de estados [19] todas las réplicas ejecutan la misma secuencia de acciones y consiguen exactamente el mismo estado. Por tanto, la gestión de aplicaciones distribuidas en la parte del servidor está siempre disponible de forma continua: si una réplica falla, el resto de las réplicas proporcionan el servicio.

En este enfoque, cuando el grupo servidor recibe una petición, todas las réplicas ejecutan el mismo código y atraviesan los mismos estados y en el mismo orden. De esta forma, cuando una operación de ejecución se envía al servidor, cada réplica realiza la tarea que le corresponda sobre una aplicación distribuida. Esto es, si el servidor tiene N réplicas y el cliente envía una operación de arranque de una aplicación distribuida, la aplicación se arranca N veces. Para evitar este problema, las peticiones que realice el grupo servidor del EPDA a otros servicios (ejecución remota, tiempo, etc.) se debe ejecutar una sola vez. Esto se consigue mediante la comunicación grupo-proceso (o comunicación N-a-1) proporcionada por Group_IO: los mensajes enviados por las N réplicas se filtran para que sólo se envíe una copia. En este enfoque el servidor replicado delega las operaciones de ejecución en un único proceso que no pertenece al grupo.

Para evitar que el proceso encargado de las operaciones de ejecución constituya un único punto de fallo, existen varios procesos, cada uno situado en un nodo físico del sistema distribuido. Llamemos a cada uno de estos procesos *proxy*. Este proxy es encarga de realizar las operaciones de ejecución sobre un componente en su nodo físico. Por ejemplo, el sistema distribuido de la figura 6 que está formado por cuatro nodos físicos debe tener cuatro proxies, uno por cada máquina (el usuario debe instalarlos cuando instala el EPDA). Así, *P1* es el proxy del EPDA que se encarga de las operaciones de ejecución en la máquina *my_pc*, *P2* controla las operaciones de ejecución en la máquina *your_pc* y así sucesivamente. En la figura 6, el grupo servidor re-envía una operación de ejecución al proxy *P3*, mediante una comunicación N-a-1, para ejecutar un componente en *her_pc*. Este proxy arranca el componente y retorna el resultado al grupo de réplicas EPDA mediante una comunicación proceso-grupo (comunicación 1-a-N).

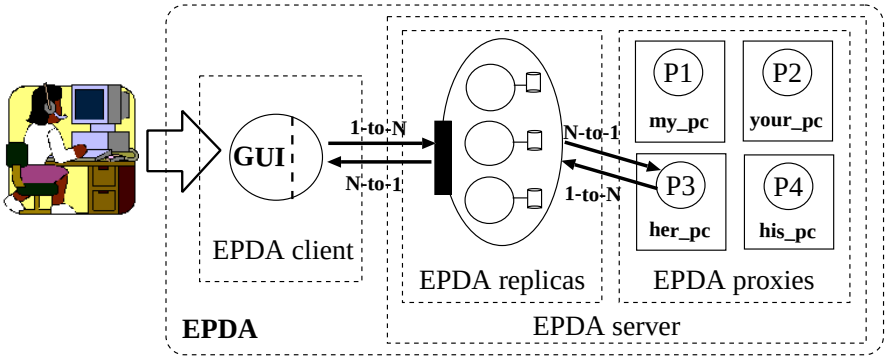


Fig. 6. EPDA robusto

Por tanto, el cliente envía cualquier petición al grupo de réplicas EPDA mediante una comunicación proceso-grupo (ver en figura como 6 as 1-a-N). Las réplicas del EPDA servidor re-envían cualquier operación de ejecución, con la información necesaria para ello, al proxy correspondiente mediante una comunicación grupo-proceso (N-a-1). Cuando el proxy lleve a cabo la operación notifica el estado de la operación al servidor (comunicación 1-a-N). Finalmente, el grupo de réplicas EPDA notifica el resultado de la operación al cliente mediante una comunicación grupo-proceso (N-a-1). En este enfoque cada réplica del grupo servidor conoce el estado de las operaciones de ejecución que han sido llevadas a cabo por el proxy[1].

Aunque el proxy permite que el grupo de réplicas se comporte como un único proceso, en algunas situaciones las réplicas pueden cooperar para obtener soluciones eficientes en la gestión del sistema distribuido. Cada réplica puede ejecutar

[1] Aunque funcionalmente este proxy se parece al demonio PVM (*pvmd*) [20], nuestro proxy es diferente porque no mantiene información acerca del resto de los proxies del sistema distribuido, que se mantiene en la base de datos de la réplica.

una porción de la operación y enviar los resultados a otras réplicas (cuando sea necesario). Por ejemplo, si un componente debe arrancarse en todos o alguno de los nodos donde las réplicas EPDA se ejecutan y permitimos que sea la réplica la que arranque a un componente (y no el proxy), la comunicación grupo-proceso no se necesita. Esta solución puede extenderse a cualquier operación que invoque a un servicio remoto (ejecución remota, tiempo, etc.).

En definitiva, en nuestra solución los servicios externos los proporciona un proxy. Sin embargo, cuando un servicio debe ejecutarse en todos o algunos nodos donde las réplicas del EPDA servidor se ejecutan, este trabajo puede realizarlo la réplica implicada.

6 EPDA-Modeller: herramienta para personalizar EPDAs

En las secciones previas se ha propuesto una arquitectura para un EPDA que resulta adecuada tanto para el paradigma de RPC como el de grupos. Estos paradigmas se han analizado mediante las plataformas Glade y Group_IO. En el análisis de los EPDAs para estas plataformas hemos observado que existen elementos comunes en esos paradigmas. Por ejemplo, Glade tiene particiones y Group_IO posee agentes, cada uno con sus propiedades específicas. En cualquiera de las plataformas estudiadas se puede utilizar una estructura en árbol para representar las relaciones entre los elementos de la aplicación distribuida; además, las operaciones de ejecución del EPDA las proporcionan los proxies localizados en las diferentes máquinas del sistema distribuido. Por tanto, independientemente de la plataforma existen un conjunto de elementos comunes a cualquier EPDA que llamaremos *EPDA-core*.

Nuestro objetivo consiste en diseñar una herramienta que nos permita personalizar el EPDA-core a una plataforma de programación distribuida concreta. La herramienta en cuestión la llamaremos EPDA-Modeller (ver figura 7). Los tipos de componentes permitidos, sus propiedades e interconexiones constituyen los elementos específicos que se necesitan para personalizar el EPDA-core. Aunque el EPDA-core proporciona algunas operaciones de gestión por omisión, se pueden agregar nuevas operaciones o bien personalizar las ya existentes. Por ejemplo, las operaciones de ejecución son comunes a cualquier plataforma pero su comportamiento puede variar de una plataforma a otra (en el paradigma de RPC los componentes se arrancan de forma individual, mientras que el paradigma de grupos se necesita arrancar todos los miembros de un grupo).

De la misma forma que la arquitectura del EPDA, el EPDA-core sigue el modelo cliente/servidor. El EPDA-core cliente se compone de una GUI similar a la GUI del EPDA cliente, sin las características específicas de la plataforma, y el código básico que proporciona la comunicación entre el cliente y el grupo de réplicas (servidor). La GUI del EPDA-core cliente se compone de las secciones de aplicaciones y nodos con sus correspondientes barras de menú. En ambas secciones existe un área reservada para representar la jerarquía de componentes mediante una estructura en árbol (parte izquierda), y otra área reservada para la edición de las propiedades específicas de los componentes que forman la plata-

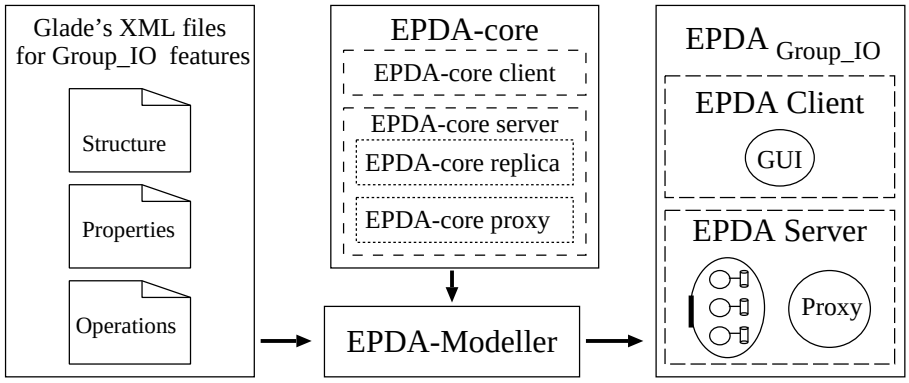


Fig. 7. Herramienta EPDA-Modeller

forma distribuida (parte derecha). La barra de menú de la GUI del EPDA-core cliente incluye operaciones como la construcción, configuración y monitorización de aplicaciones distribuidas (las operaciones que implican cambios en la estructura de la aplicación durante su ejecución no se encuentran en todas las plataformas distribuidas). La GUI a medida del EPDA cliente permite crear ejemplares de componentes (sólo de aquellos que estén permitidos), introducir las propiedades específicas de cada componente, establecer las relaciones entre componentes sin violar las reglas de relación, y presentar al usuario aquellas operaciones de gestión que se encuentran disponibles para la plataforma distribuida.

En la parte del servidor, el EPDA-core servidor contiene el código básico que proporciona la comunicación entre el grupo de réplicas (servidor) y los clientes y proxies: por un lado, el EPDA-core de la réplica proporciona la comunicación básica entre el grupo de réplicas y el cliente, además de la comunicación básica entre el grupo de réplicas y el proxy; por otro lado, el EPDA-core del proxy contiene el código de comunicación básico que permite la comunicación entre el proxy y el grupo de réplicas. El EPDA-Modeller utiliza la estructura de la aplicación y las propiedades de sus componentes, proporcionadas por el usuario, para crear una base de datos a medida y una API (*Application Programming Interface*) que contiene un conjunto de funciones a medida para acceder a la base de datos. Por cada operación de ejecución específica, el usuario proporciona el código específico del proxy, además de la información del componente que el EPDA-Modeller necesita para construir el código de la réplica para esa operación. El EPDA-Modeller adapta el EPDA-core del proxy y de la réplica a partir de la información proporcionada por el usuario (estructura, propiedades y operaciones) para conseguir un EPDA a medida para la plataforma correspondiente. Después de la adaptación, el servidor y su base de datos deben instalarse. Para una instalación robusta se deben instalar diferentes servidores con su base de datos en diferentes nodos del sistema distribuido. Para el soporte de las operaciones de ejecución en los diferentes nodos del sistema distribuido, en cada nodo se debe instalar el proxy a medida generado por el EPDA-Modeller.

La parte gráfica del EPDA-core cliente se ha programado mediante la herramienta *GUI Builder Glade*[2] y GtkAda [21]. La parte de comunicaciones tanto del EPDA-core cliente como del servidor se ha implementado mediante la biblioteca de comunicaciones Group_IO. Además, el EPDA-core servidor utiliza un *binding* a la base de datos postgresql. El usuario proporciona al EPDA-Modeller las características específicas de una plataforma de programación distribuida (su estructura, sus propiedades y las operaciones de ejecución) a través de tres archivos en formato XML, generados mediante la herramienta *GUI Builder Glade*. El EPDA-Modeller recorre dichos archivos y extrae la información necesaria para personalizar el EPDA-core cliente y servidor.

7 Ejemplo de uso

Esta sección describe un ejemplo de uso de la herramienta EPDA-Modeller para obtener un EPDA a medida para las aplicaciones distribuidas construidas con la biblioteca de comunicaciones Group_IO. El usuario proporciona al EPDA-Modeller los tipos de componentes permitidos en Group_IO (aplicaciones, grupos y agentes) con sus propiedades y las reglas de relación entre ellos. Toda esta información se proporciona al EPDA-Modeller mediante dos archivos en formato XML. Supongamos que en el EPDA a medida deseamos que aparezca la operación de instalación. El usuario debe proporcionar al EPDA-Modeller el nombre de la operación, el código que proporciona la funcionalidad en el proxy y la información necesaria para la funcionalidad de la réplica. Un ejemplo de proxy puede recibir el mensaje del servidor con los siguientes datos: los recursos necesarios para instalar el agente y la localización de los ejecutables. A partir de esa información el proxy comprueba si su nodo posee los requisitos necesarios para instalar el agente y si su sistema operativo coincide con el del ejecutable del agente. Para que el EPDA-Modeller añada la operación de instalación al código de la réplica, el usuario debe proporcionar un archivo en formato XML con las propiedades que el servidor necesita para enviar al proxy la operación de instalación: nodo destino del agente, fuente del agente (nodo y ejecutable) y los recursos que se necesitan para instalar el agente.

El EPDA-Modeller proporciona un EPDA para Group_IO, cuyos componentes permitidos son las aplicaciones, los agentes y los grupos. El EPDA evita que el usuario viole las reglas de relación entre los componentes permitidos y la única operación que aparece en la barra de menú de la GUI es la correspondiente a la instalación. Inicialmente, el usuario genera ejemplares de los componentes permitidos. Para cada componente se deben introducir sus propiedades a través del formulario en la parte derecha de la GUI. Cuando se confirmen las propiedades, el componente, sus propiedades y la relación con otros ejemplares de la aplicación se envía al servidor que lo almacena en su base de datos. Para instalar uno de los ejemplares de los agentes generados, el usuario lo selecciona en la GUI y accede a la operación de instalación en la barra de menú. Esa orden se envía al

[2] Esta herramienta no es la misma que el Glade proporcionado con GNAT para sistemas distribuidos.

servidor que extrae de la base de datos la información que el proxy necesita del agente. A continuación, el servidor redirige la operación de instalación con los parámetros extraídos de la base de datos (fuente del agente y recursos necesarios para su instalación) al proxy correspondiente (la asociación entre proxies y nodos físicos se almacenan en la base de datos de las réplicas). Finalmente, si todo va bien se instala el agente, el proxy informa de ello al grupo de réplicas, que a su vez lo notifica al cliente; si se produjo un error se le notificará al servidor que lo indicará al cliente.

8 Conclusiones y trabajo actual

Se ha presentado un entorno para programar aplicaciones distribuidas (EPDA) que soporta las operaciones de construcción, configuración, ejecución, monitorización y evolución de aplicaciones distribuidas. El entorno se puede instalar como robusto para garantizar la continuidad de su servicio en presencia de fallos en los nodos. La arquitectura propuesta para el EPDA resulta adecuada para plataformas distribuidas como Glade y Group_IO. Un análisis de los EPDAs para estas plataformas nos ha permitido extraer aquellos elementos que son comunes a cualquier plataforma, que constituyen el EPDA-core. La herramienta de modelado construye un EPDA a medida para diferentes plataformas distribuidas. Ello permite que si el programador cambia de plataforma sólo tenga que familiarizarse con las características específicas de la plataforma, el resto del EPDA es el mismo.

Aunque se ha comprobado el EPDA para Glade y la biblioteca de comunicaciones Group_IO, pensamos que el EPDA se puede utilizar para otras plataformas como Ensemble, Electra, Transis, Totem, etc. Actualmente nos encontramos evaluando el rendimiento del proxy a medida para diferentes operaciones de gestión de aplicaciones distribuidas e incluso en diferentes situaciones, como cuando el nodo del proxy y el de la réplica coinciden.

References

1. J. Kramer. Configuration Programming - A Framework for the Development of Distributable Systems. In *Proceedings of the Int. Conference on Computer Systems and Software Engineering (Compeuro 90)*, pages 374–384, Israel, May 1990.
2. M. Hayden. *The Ensemble System*. Faculty of the Graduate School of Cornell University, 1998. Doctoral Dissertation.
3. D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(1):65–70, April 1996.
4. F. Guerra, J. Miranda, A. Álvarez, and S. Arévalo. An Ada Library to Program Fault-Tolerant Distributed Applications. In *1997 Ada-Europe International Conference on Reliable Software Technologies*, pages 230–243, London, June 1997. Springer-Verlag.
5. M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. Durra: a Structure Description Language for Developing Distributed Applications. *Software Engineering Journal*, 8(2):83–94, March 1993.

6. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.
7. L. Pautet and S. Tardieu. *GLADE User's Guide. Version 3.13p*. GNAT, May 2001.
8. K. Ng, J. Kramer, J. Magee, and N. Dulay. The Software Architect 's Assistant - A Visual Environment for Distributed Programming. In *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, January 1995.
9. Gargaro, Kermarrec, Nana, Pautet, Smith, Tardieu, Theriault, Volz, and Waldrop. *Adept (Ada 95 Distributed Execution and Partitioning Toolset)*. Technical Report, April 1996.
10. G. Etzkorn. Change Programming in Distributed Systems. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 140–151, United Kingdom, March 1992.
11. L. Pautet, T. Quinot, and S. Tardieu. Building Modern Distributed Systems. In *2001 Ada-Europe International Conference on Reliable Software Technologies*, pages 122–135, Belgium, May 2001. Springer-Verlag.
12. T. Batista and N. Rodriguez. Dynamic Reconfiguration of Component-based Applications. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 32–39, Ireland, May 2000.
13. V.C. Paula, G.R. Ribeiro, and P.R.F. Cunha. Specifying and Verifying Reconfigurable Software Architectures. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 21–31, Ireland, May 2000.
14. S. Maffeis. Piranha:A CORBA Tool for High Availability. *IEEE Computer*, pages 59–66, April 1997.
15. Intermetrics. *Ada 95 Language Reference Manual.* International Standard ANSI/ISO/IEC-8652: 1995, January 1995.
16. K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.
17. F. Guerra, J. Miranda, J. Santos, E. Martel, L. Hernández, and E. Pulido. Programming Systems with Group_io. In *Proceedings of the Tenth Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 188–195, Canary Islands, Spain, January 2002. IEEE Computer Society Press.
18. F. Guerra, J. Miranda, Santos J., and J. Calero. Building Robust Applications by Reusing Non-Robust Legacy Software. In *2001 Ada-Europe International Conference on Reliable Software Technologies*, pages 148–159, Belgium, May 2001. Springer-Verlag.
19. F. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
20. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderman. *Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* ISBN:0-262-57108-0. The MIT Press, 1994.
21. E. Briot, J. Brobecker, and A. Charlet. *GtkAda Reference Manual. Version 1.2.12.* ACT Europe, March 2001.

Java: la concurrencia y otras pifias

Pedro de las Heras Quirós, José Centeno González, Jesús M. González Barahona, and Vicente Matellán Olivera

Universidad Rey Juan Carlos, C/ Tulipán S/N, Móstoles E-28933
{pheras,jcenteno}@gsyc.escet.urjc.es

"Java evolved out of a Sun research project started six years ago to look into distributed control of consumer electronics devices. It was not an academic research project studying programming languages: Doing language research was actively an antigoal"

James Gosling, 1997, en [Gos97]

Resumen En este artículo se exponen aspectos del lenguaje Java que pueden hacerlo poco adecuado tanto como herramienta para enseñar a programar como para desarrollar grandes proyectos de programación. En algunos casos se trata de ausencias de mecanismos en el lenguaje, en otros de mecanismos propensos a ser mal utilizados, y en otros de verdaderas "bombas de relojería", como la introducción de nuevos elementos del lenguaje camuflados como pequeñas modificaciones (clases internas), o el mal uso de terminología bien establecida desde hace más de veinticinco años (monitores).

1 Introducción

La tecnología Java (término con el que se puede englobar a la máquina virtual JVM, a las bibliotecas y al lenguaje de programación) ha experimentado una fuerte expansión desde su aparición en el año 1995, en el número de programadores, de bibliotecas más o menos estándar, de anuncios en medios de comunicación, de libros, de páginas de WWW, de modificaciones y extensiones de las bibliotecas y hasta del propio lenguaje.

La tecnología Java es interesante fundamentalmente por haber difundido a gran escala elementos tecnológicos ya existentes, pero poco utilizados en la industria. Por ejemplo, la recolección automática de basura, la transportabilidad del código entre diferentes plataformas o la programación concurrente con soporte del lenguaje, por citar sólo algunos.

Este artículo está estructurado en secciones independientes, que pueden ser leídas en cualquier orden. En cada una de ellas se exponen algunas críticas a aspectos del lenguaje que pueden hacerlo poco adecuado tanto como herramienta para enseñar a programar como para desarrollar grandes proyectos de programación. En la primera sección se exponen aspectos relacionados con la concurrencia, después se analiza la inicialización de clases y objetos, el despacho dinámico, las clases internas, y la ausencia de genéricos. Concluye el artículo con algunas notas sobre la evolución y el control de la tecnología Java.

2 Aspectos relacionados con la concurrencia

En [BH99] Per Brinch Hansen examina los mecanismos de sincronización de Java, y afirma que no es cierto que Java soporte monitores. Refiriéndose a lo que en Java se denominan monitores, Brinch Hansen dice: "*... It is just a programming style that imitates insecure monitors. Almost any programming language (including assembly language) enables you to adopt programming styles based on abstract concepts that are not supported directly by the language*".

En esta sección se exponen algunos de los aspectos negativos relacionados con la concurrencia en Java.

La interfaz de la clase Thread

El método `run` de la clase `Thread` está definido como `public`. Cuando se extiende esta clase y se reemplaza el método `run` no se puede restringir su visibilidad haciéndolo, por ejemplo, `private`.

Por lo tanto el método `run` de un objeto *thread* puede ser llamado desde cualquier parte del código que tenga visibilidad de la clase a la que pertenece el objeto *thread*. Esto no es lógico, pues este método está pensado para que sea llamado por el sistema, una vez que se activa el *thread* asociado a la clase.

Dispersión del código sincronizado

Las sentencias `synchronized` se introdujeron en Java para poder utilizar el monitor asociado a un objeto de una clase que no tiene métodos `synchronized`:

```
synchronized (o) {
 // código protegido
}
```

Esta construcción puede utilizarse también con objetos de clases que sí tienen métodos `synchronized`. Por lo tanto, el código de sincronización ligado a un monitor puede aparecer disperso en el código. No basta con ver el código de los métodos `synchronized`, sino que hay que buscar todas las sentencias `synchronized (o)`. En caso de existir alias del objeto la búsqueda puede resultar aún más complicada:

```
p = o;

synchronized (p) {
  // ... No puede haber otro thread ejecutando código más que yo
  p.notify ();
}

// ...
```

Los monitores de Java no tienen variables condición

En otros lenguajes de programación los monitores se utilizan junto con el mecanismo de las variables condición. Cada variable condición puede utilizarse para que se bloqueen en ella los *threads* que no cumplen una condición determinada. A la hora de llamar a `wait()` (o la sentencia equivalente en cada lenguaje) se especifica en qué variable condición se quiere quedar bloqueado el *thread* llamante. Cuando la condición por la que esperaban los *threads* que están bloqueados en una variable condición se cumple, otro *thread* puede despertarlos: al llamar a `notify()` (o la sentencia equivalente en cada lenguaje) se especifica la variable condición.

En Java sólo existe una variable condición asociada a cada monitor, por lo que al llamar a `wait()` o a `notify()` no hace falta explicitar su nombre (de hecho no se puede referir en el código).

En esa misma variable condición puede haber varios *threads* bloqueados, cada uno de ellos esperando por condiciones distintas. Cuando otro *thread* compruebe que alguna de las condiciones se cumple puede despertar a alguno de los *threads* bloqueados ejecutando `notify()`. Pero puede ocurrir que el *thread* despertado no esté esperando por esa condición. Para paliar este problema en Java se utiliza el método `notifyAll()` en lugar de `notify()`. Llamando a `notifyAll()` el *thread* llamante despierta a todos los *threads*. Se está por tanto gastando un tiempo innecesario, pues cada uno de los *threads* despertados tiene que ejecutar código para comprobar si es su condición la que se cumple, y si no volver a bloquearse llamando a `wait()`. Además no está acotado el tiempo que puede tardar en entrar el *thread* para el que se cumple la condición.

En lugar de utilizar `notifyAll()` se pueden simular las variables condición existentes en otros lenguajes de programación utilizando un monitor extra para cada una de ellas (ver [Lea96]), además del "verdadero monitor":

```
class VariableCondición { Boolean deboDormir = false; };
class VerdaderoMonitor {
  VariableCondición c1 = new VariableCondición ();
  VariableCondición c2 = new VariableCondición ();
  public void unMetodo () {
    // Adquirimos primero el monitor de la variable condición
    synchronized (c1) {
      // Ahora adquirimos el monitor
      synchronized (this) {
        // Sólo ahora estamos realmente dentro del monitor,
        // y podemos consultar o actualizar el estado protegido
        if (condicion2) c2.notify ();
        if (not condicion1) c1.deboDormir = true;
      }
      if (c1.deboDormir) c1.wait ();
    }
  }
}
```

Como se puede apreciar en el ejemplo, los *threads* nunca se quedan bloqueados dentro del "verdadero monitor", sino dentro del monitor de la variable condición (cuando se ejecuta `c1.wait()` en el ejemplo). Sin embargo, la condición hay que comprobarla cuando está dentro del "verdadero monitor", pues se necesita que otros *threads* no estén modificando el estado mientras que se comprueba la condición. Por ello se anota en la variable `deboDormir` si la condición se cumplió o no. Si la llamada a `c1.wait()` se hiciese cuando está dentro del "verdadero monitor", ningún otro *thread* podría pasar al "verdadero monitor" a despertarnos.

Al tener ahora que adquirir siempre dos monitores, esta solución puede plantear problemas de rendimiento. A cambio se obtiene la ventaja de poder despertar a un *thread* que espera por una condición determinada, sin despertar a los demás como ocurre cuando se utiliza `notifyAll()` (en el ejemplo, `c2.notify()`).

Al no estar soportadas en el lenguaje las variables condición hay que recurrir a técnicas como ésta, cuyo uso no es sencillo, pudiendo tener consecuencias catastróficas en caso de usarse mal (interbloqueos, inanición,...).

En [OW99] (pp. 98) se muestra otra implementación de variables condición.

Falta de equidad en el acceso a monitores

Los *threads* que son despertados de un `wait()` tienen que competir por la adquisición del monitor con otros *threads* que en ese momento estuviesen intentando acceder a través de métodos o sentencias `synchronized`.

Esto hace que la implementación de políticas equitativas de acceso al monitor no sea inmediata. En otros lenguajes de programación que tienen monitores, cuando se despierta a un *thread* que ya estaba esperando en un `wait()`, éste tiene prioridad sobre los que esperan para entrar en el monitor, no teniendo que competir con ellos para adquirir el monitor.

Mezcla de métodos sincronizados y no sincronizados

En Java se pueden mezclar en una misma clase métodos `synchronized` con otros que no lo son. En las instancias de este tipo de clases puede haber varios *threads* ejecutando código a la vez: si bien sólo uno de ellos puede haber entrado[1] a través de alguno de los métodos o sentencias `synchronized`, a través del resto de métodos podría haber entrado un número cualquiera de *threads*:

[1] Entrar en el monitor equivale a adquirir la exclusión mutua del monitor asociado a la instancia de la clase

```
class UnMonitor {
  synchronized void unMetodoSynchronized () {
    while (!unaCondicion) { try {wait(); }
                            catch (InterruptedException e) { return; }
    } // ... Puede haber otros threads ejecutando código
  }

  void unMetodoNormal () {
    // ... Puede haber otros threads ejecutando código
  }
}
```

Usualmente el mecanismo del monitor en otros lenguajes no permite esta peligrosa flexibilidad. Utilizando la terminología de Java, en estos lenguajes todos los métodos de un monitor son `synchronized`.

La flexibilidad que aporta Java al permitir la mezcla de métodos `synchronized` con otros que no lo son puede ser fuente de frecuentes errores de programación. Una de las ventajas de los monitores respecto a otros mecanismos de sincronización como los semáforos es el mayor nivel de abstracción que proporcionan aquéllos. La exclusión mutua se obtiene de manera implícita al utilizar un monitor, mientras que si se utilizan semáforos hay que implementarla explícitamente, por lo que aumenta la probabilidad de equivocarse al programar. La posibilidad de mezclar métodos `synchronized` con otros que no lo son convierten a los monitores de Java en un mecanismo a medio camino entre los semáforos y los monitores de otros lenguajes, en lo que a posibilidad de cometer errores por parte del programador se refiere.

Podría pensarse que esta flexibilidad permite implementar esquemas de sincronización útiles como lectores/escritor etc. Sin embargo no es así: si hay dentro del monitor *threads* (lectores) que han entrado a través de métodos no `synchronized`, no se prohibe la entrada a un nuevo *thread* (escritor) a través de uno de los métodos `synchronized`.

Si los métodos que no son `synchronized` no se hacen públicos, o bien no se escriben métodos no `synchronized` en una clase que se vaya a utilizar como monitor, este problema desaparece. Pero cualquiera de estas soluciones requiere una disciplina por parte del programador, no contando con la ayuda del lenguaje.

Un problema similar afecta también al monitor que cada clase tiene asociado en Java. Pueden utilizarse métodos *static synchronized* o sentencias`synchronized (getClass())` para acceder a datos protegidos por el monitor de la clase. Pero nada impide acceder a esos mismos datos sin sincronizarse en el monitor de la clase.

3 Bloques de inicialización

La creación y destrucción de los objetos no primitivos se controla en Java mediante los constructores, los bloques de inicialización y el método `finalize`. Por razones de espacio nos limitaremos a comentar algunos aspectos de los bloques de inicialización.

Bloques de inicialización estáticos y de instancia

Los bloques de inicialización estáticos o de clase sirven para inicializar el estado de una clase. Puede haber varios de ellos en una misma clase, ejecutándose éstos según el orden en el que aparecen en el código fuente, en el momento en que se carga la clase.

Los bloques de inicialización no estáticos o de instancia (iguales pero sin la palabra `static`) se introdujeron junto con las clases internas anónimas en la versión 1.1 de la especificación de Java (ver el apartado 5).

Pero una vez introducidos en el lenguaje se pueden utilizar también en cualquier clase. Estos bloques se ejecutan cada vez que se crea una instancia de la clase a la que pertenecen, antes de ejecutarse su constructor, y después de ejecutarse el constructor de la clase padre. De haber varios, se ejecutan según el orden en el que aparecen en el código fuente.

Como el lector puede apreciar, la función de los bloques de inicialización no estáticos se solapa con la función de los constructores.

El uso combinado de bloques de inicialización estáticos y no estáticos puede producir efectos no deseados, como podemos ver en el siguiente programa:

```
class T1 {
  {
    System.out.println ("  Primer bloque de inic. de instancia");
  }

  static {
    System.out.println ("Primer bloque de inic. estático");
    T1 b = new T1 ("b");
  }

  {
    System.out.println ("  Segundo bloque de inic. de instancia");
  }

  static {
    System.out.println ("Segundo bloque de inic. estático");
  }

  public T1 (String name){
    System.out.println ( "    Constructor de: " + name);
  }
}

public class Main {
  static public void main (String [] argv) {
        T1 a = new T1("a");
  }
}
```

En la clase `T1` se han incluido 4 bloques de inicialización, dos estáticos y dos de instancia. En el método `main` de la clase `Main` creamos una instancia de la

clase `T1`. A partir de ese momento se carga la clase `T1`, y comienza a ejecutarse el primer bloque de inicialización estático. Hay otro bloque de inicialización estático, pero como se puede ver en la salida del programa por pantalla que se muestra a continuación, ¡antes de que se ejecute el segundo bloque de inicialización estático se llega a crear una instancia de T1!:

```
Primer bloque de inic. estático
  Primer bloque de inic. de instancia
  Segundo bloque de inic. de instancia
    Constructor de: b
Segundo bloque de inic. estático
  Primer bloque de inic. de instancia
  Segundo bloque de inic. de instancia
    Constructor de: a
```

Se trata de la instancia `b`, que se declara en el primer bloque de inicialización estático. En la salida del programa por pantalla podemos apreciar cómo en ese momento se ejecutan los dos bloques de inicialización de la instancia `b`, y a continuación su constructor. Nótese que la clase aún no está inicializada (falta el segundo bloque estático) y, sin embargo, ya se ha creado una instancia de ella.

Más tarde se ejecuta por fin el segundo bloque de inicialización estático, para terminar con la ejecución de los bloques de inicialización de la instancia `a` y de su constructor.

4 Problemas del despacho dinámico

En Java la invocación de métodos siempre utiliza despacho dinámico[2]. Esto plantea problemas cuando se extiende una clase: puede haber código que ya estaba probado y que ahora, al heredarlo, deja de funcionar, aun cuando éste no ha sido modificado. Este grave problema se presenta cuando el código heredado hace llamadas a métodos que sí se han reemplazado.

A continuación se presenta un ejemplo de este problema. Se tiene una clase `T1`, con los métodos `A` y `B`, y el atributo `i`. La clase `T2` extiende `T1`, heredando el método `A`, y reemplazando el método `B` y el atributo `i`. Además, en la implementación de `A` se llama a `B`:

```
class T1 {
  int i = 1;
  public void A (){
    System.out.println (``i='' + i);
    B ();
  }
  public void B (){
    System.out.println
      (``En B de T1'');
  }
}
```

```
class T2 extends T1 {
  int i = 2;
  public void B () {
    System.out.println
      (``En B de T2'');
  }
}
```

[2] Salvo cuando se utiliza *super()*

En el método `main` de la clase `Main` se declaran dos objetos, `t1` y `t2`, de tipos `T1` y `T2` respectivamente. Luego se llama al método `A` de ambos objetos:

```
public class Main {
  static public void main (String [] argv ) {
    T1 t1 = new T1 ();
    T2 t2 = new T2 ();

    System.out.println (``>>>> t1.A()'');
    t1.A();
    System.out.println (``\n'' + ``>>>> t2.A()'');
    t2.A();
  }
}
```

Al ejecutar este programa se obtiene la siguiente salida en la pantalla:

```
>>>> t1.A()
i = 1
En B de T1

>>>> t2.A()
i = 2
En B de T2
```

Como en Java todas las llamadas despachan dinámicamente (salvo `super`), la llamada `t2.A()` acaba invocando al método `B` del tipo `T2`, y no al método `B` del tipo `T1`. En el caso del atributo `i`, en la llamada `t2.A()` se obtiene en la pantalla el valor 2, al reemplazarse el atributo de `T1` por el proporcionado por `T2`.

¿Cómo se puede forzar a que la llamada que se hace dentro de `A` quede ligada al método `B` de la clase `T1` en lugar de despachar dinámicamente? ¿Y si se quiere que el atributo `i` que se imprime en el método `A` sea siempre el de la clase `T1`?

Para responder a ambas preguntas se modifica el método `A`, para forzar la conversión a tipo `T1` del parámetro implícito `this` tanto en la llamada a `B` como en el acceso al atributo `i`:

```
class T1 {
  int i = 1;

  public void A (){
    System.out.println (``i='' + ((T1)this).i);
    ((T1)this).B ();
  }
  public void B (){
    System.out.println
      (``En B de T1'');
  }
}
```

La salida obtenida ahora en pantalla es la siguiente:

```
>>>> t1.A()
i = 1
En B de T1

>>>> t2.A()
i = 1
En B de T2
```

Como puede apreciarse, en el caso del atributo `i` la conversión del tipo de `this` sirve para conseguir el efecto buscado: en nuestro ejemplo, la llamada `t2.A()` acaba imprimiendo el valor 1 del campo `i` del tipo `T1`, como era de esperar.

Pero, sorprendentemente, con los métodos el comportamiento es distinto: a pesar de la conversión explícita de `this`, la llamada a `((T1)this).B()` dentro de `A` despacha dinámicamente, y es el método `B` de `T2` el que se ejecuta.

Este problema es grave, pues complica enormemente la reutilización del código:

- El programador que creó `T1` con los métodos `A` y `B` nunca puede diseñar `A` de forma que la llamada a `B` funcione adecuadamente para cualquier posible reemplazamiento que se haga en una futura extensión de `T1`. Tampoco podría escribir un código que forzara a que esa llamada a `B` no despachara nunca[3].
- El programador que cree `T2` extendiendo `T1` puede que no disponga de la implementación de `T1`, por lo que puede resultarle difícil saber que el código de `A` llama a `B`, y que puede dejar de funcionar si se reemplaza dicha `B` en `T2`.

Para realizar un *software* que no padezca estos problemas en Java hay dos alternativas:

- Declarar la clase `T1` o el método `B` de `T1` como `final`, de forma que el método `B` de `T2` no pueda declararse. Esto coarta la extensibilidad del software.
- Volver a probar la biblioteca reutilizada (en este caso `T1`). Las desventajas de esta alternativa son evidentes: coste de desarrollo incrementado, posibilidad de errores por pruebas no exhaustivas, etc. Para paliar esta situación pueden emplearse soluciones no basadas en el lenguaje: el programador de `T1` debería documentar que en `A` se realiza una llamada al método `B`, o bien entregar el código fuente de su biblioteca.

5 Las clases internas

Las clases internas fueron introducidas en Java a partir de la versión 1.1. Se trata de proporcionar una construcción que facilite la codificación en todas aquellas situaciones en las que se precise pasar como parámetro un método a otro para

[3] Nótese que en C++ las llamadas a los métodos se despachan dinámicamente por defecto, pero puede forzarse a que no lo hagan.

que este último haga algo invocándolo. Este problema se resuelve típicamente con punteros a funciones en lenguajes como C, C++ o Ada. En cambio, y dado que en Java un método no es un un objeto de un tipo del lenguaje (ni primitivo ni referenciado), la solución adoptada en este lenguaje pasa por la creación de una clase auxiliar con el método en cuestión y pasar esta clase como parámetro. Podría haberse adoptado alguna solución alternativa como la propuesta por Pizza[4].

Al utilizar clases cuyo único objetivo es ser contenedores de los métodos que se quieren pasar como parámetro, el código se vuelve oscuro: la farragosa sintaxis necesaria en Java 1.0 y la proliferación por el código de estas clases auxiliares mezcladas con las clases normales llevó a los diseñadores del lenguaje a introducir las clases internas.

Las clases internas facilitan, en comparación con la situación previa a Java 1.1, la escritura de programas que requieran la funcionalidad proporcionada por los inexistentes punteros a métodos. Pero la introducción de esta construcción ha tenido su precio:

- Se ha necesitado realizar nuevos cambios[5] en otros aspectos del lenguaje para eliminar problemas o ambigüedades con el uso de las clases internas. Ninguno de dichos cambios era necesario antes de la introducción de las clases internas, con lo que se complica innecesariamente el lenguaje.
- Las posibles utilizaciones sintácticamente correctas de las clases internas permiten escribir código poco intuitivo y escasamente legible.

Por razones de espacio nos limitamos a exponer a continuación algunos ejemplos de usos de las clases internas que producen código poco intuitivo y escasamente legible.

Construcciones peculiares

Las clases internas introducen el concepto de "inclusión" de manera independiente y ortogonal al de "herencia". Así, además de una jerarquía de clases ordenadas por relaciones de herencia aparece ahora otra jerarquía de clases incluidas unas dentro de otras. Ambas jerarquías pueden además entremezclarse a voluntad.

Esto conduce a la posibilidad de escribir código perfectamente legal pero de oscuro significado y comportamiento. A continuación se muestran algunos ejemplos:

[4] Pizza es una extensión a Java que proporciona funciones similares a los bloques de Smalltalk, que pueden ser pasadas como parámetros de métodos, almacenadas en variables, o devueltas como resultado de otra función. Pizza aporta también tipos genéricos, otra carencia notable de Java. Pizza está disponible en http://www.cis.unisa.edu.au/~pizza/.

[5] http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html

1. Clase contenedor que contiene como miembro a una clase interna suya

```
public class A {
  int j = 1;
  B b = new B ();

  public class B {
    int k = 2;

    public void printValues () {
      System.out.println(k);
      System.out.println(this.k);      // lo mismo
      System.out.println(A.this.b.k);  // lo mismo
      System.out.println(j);           // acceso a j por inclusión
//    System.out.println(this.j);      // ilegal, B no tiene una j
      System.out.println(A.this.j);    // acceso a j por inclusión
    }
  }
}
```

2. Clase interna que extiende a una clase de nivel máximo distinta de su clase contenedor

```
public class A {
  int j = 1;
  B b = new B ();

  public class B extends C {
    int k = 2;

    public void printValues () {
      System.out.println(k);
//    System.out.println(j);        // ilegal, hay que explicitar
      System.out.println(this.j);   // acceso a la j heredada de C
      System.out.println(C.this.j); // lo mismo, es legal
      System.out.println(A.this.j); // acceso a la j de A
    }
  }
}

public class C {
  int j = 3;
}
```

3. Clase interna que extiende a su propia clase contenedor
   Este caso es especialmente peculiar. Se propone al lector que antes de continuar intente predecir cuál debe ser la salida del siguiente programa:

```
public class A {
  public int j = 1;
  public int k = 0;

  public class B extends A {
    public void printValues () {
      j++;
      System.out.println(j);
      System.out.println(A.this.j);
      System.out.println(A.this.k);

    }
  }
}


public class Main {
  public static void main (String [] argv )  {
    A a = new A ();
    A.B b = a.new B ();
    b.printValues ();
    System.out.println (a.j);

  }
}
```

La salida que da el programa es `2 2 0 1`. Es decir, las dos referencias a `j` en `printValues()` se refieren a la `j` heredada (que se incrementa en 1), mientras que el `1` final corresponde a la `j` de la instancia contenedor, que aún sigue valiendo 1.
Nótese lo sorprendentes que resultan los siguientes aspectos:

- no es posible referirse en `printValues()` a la `j` de la instancia contenedor
- la sintaxis `A.this.k` sí se refiere a la `k` de la instancia contenedor, mientras que `A.this.j` hace referencia a la `j` heredada en B

6 Parametrización en tiempo de compilación: ausencia de genéricos

Una de las ausencias notables de Java es el soporte de genéricos, presentes en lenguajes como C++ (*templates*) o Ada (genéricos). Los genéricos son útiles para realizar bibliotecas de componentes reusables, como contenedores de elementos de diversos tipos.

En Java la ausencia de genéricos se puede suplir utilizando contenedores (como vectores o listas) cuyos elementos son de un tipo básico (generalmente Object), de forma que cualquier elemento que sea de un subtipo de aquél pueda almacenarse en el contenedor. Veamos un ejemplo:

```
class CartapacioGenerico {
  public void mete (Object x) {
   // ...
  }

  public Object saca() {
    // ...
  }
}

class Main {
  public static void main (String[] args) {

    // Cartapacio de bytes
    CartapacioGenerico cb = new CartapacioGenerico ();
    cb.mete (new Byte (0)); cb.mete (new Byte (1));
    Byte x = (Byte)cb.saca();

    // Cartapacio de Strings
    CartapacioGenerico cs = new CartapacioGenerico ();
    cs.mete (``cero''); cs.mete (``uno'');
    String y = (String)cs.saca();

    // Cartapacio de strings tratado como cartapacio de bytes
    Byte z = (Byte)cs.saca(); // Compila pero genera excepción
  }
}
```

Sin embargo esta solución implica múltiples promociones de tipos en tiempo de ejecución, con la consiguiente merma de rendimiento al comprobar compatibilidades de tipos. En el ejemplo, la última sentencia se compila correctamente, pero en tiempo de ejecución generará una excepción.

Con los genéricos estas comprobaciones se pueden realizar en tiempo de compilación, obteniendo programas más rápidos y robustos.

Existen múltiples propuestas para añadir genéricos a Java, y es seguro que el lenguaje acabará incorporando este útil mecanismo[6].

Una de las propuestas actuales es GJ, que extiende el lenguaje Java para incluir tipos y métodos genéricos[BOSW98] y se puede consultar en WWW[7].

[6] http://www.research.avayalabs.com/user/wadler/pizza/gj/index.html#sun-prototype

[7] http://www.cis.unisa.edu.au/~pizza/gj/

7 ¿Quién controla la tecnología Java?

La ausencia de un estándar y el oscuro proceso mediante el que se regula la evolución de la tecnología Java puede ser una de las amenazas más importantes que acechan a la comunidad y a la propia tecnología Java.

La experiencia del proceso de estandarización del lenguaje C++ (no hubo un estándar ISO hasta noviembre de 1997) debería bastar para comprender los problemas asociados a la ausencia de un estándar: múltiples versiones incompletas de elementos del lenguaje, herramientas de desarrollo continuamente modificadas, etc. El lenguaje Java ha venido padeciendo este tipo de problemas hasta la fecha: el lenguaje ha estado en constante evolución (clases internas, genéricos), por lo que no existe una definición completa, detallada y suficientemente estable de Java.

Los primeros pasos del proceso de estandarización ISO de la plataforma Java los inició Sun Microsystems, en su calidad de organización que desarrolla y posee las especificaciones públicas de la tecnología, en marzo de 1997. En 1999 Sun Microsystems propuso estandarizar la tecnología en ECMA, pero ese mismo año retiró[8] la propuesta.

El Java Community Process regula el desarrollo de la tecnología Java. Desgraciadamente el control que ejerce Sun sobre esta comunidad está poniendo muchas trabas a la participación de organizaciones ligadas al software libre en la comunidad Java[9].

Referencias

[AG98] Ken Arnold and James Gosling. *The Java Programming Language Second Edition.* Addison Wesley, 1998.

[BH99] Per Brinch Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices*, 34(4), April 1999.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98) Conference*, Vancouver, Canada, October 1998.

[Bro97] Benjamin M. Brosgol. Java for Ada Programmers. Tutorial Notes. Tri-Ada'97 Conference., November 1997.

[Fla97] David Flanagan. *Java in a Nutshell 2nd edition.* O´Reilly, 1997.

[Gos97] James Gosling. The Feel of Java. *IEEE Computer*, 1997.

[Lea96] Doug Lea. *Concurrent Programming in Java. Design Principles and Patterns.* Addison Wesley, 1996.

[OW99] Scott Oaks and Henry Wong. *Java Threads, 2nd Edition.* O'Reilly, 1999.

[Taf97] S. Tucker Taft. High Integrity Object-Oriented Programming with Ada 95. Keynote Speech. Tri-Ada'97 Conference., November 1997.

[8] http://www.javasoft.com/aboutJava/standardization/index.html

[9] http://jakarta.apache.org/site/news.html#0313.1