YA: Fast and Scalable Discovery of Idle CPUs in a P2P network

Javier Celaya and Unai Arronategui
University of Zaragoza,
Department of Computer Science and Systems Engineering
C/ María de Luna 1, Ed. Ada Byron, 50018 Zaragoza, Spain
{jcelaya, unai}@unizar.es

Abstract

Discovery of large amounts of idle CPUs in fully distributed and shared Grid systems is needed in relevant applications and is still a challenging problem. In this paper we present a fast, scalable and efficient discovery protocol founded on a tree-based peer-to-peer (p2p) network with fault-tolerant capabilities and locality features. Each system node stores a good estimation of the number of CPUs that are available in its branch. Each node notifies its father about changes in this value only when it is meaningful enough. This allows low overhead and a stable behavior with concurrent and dynamic allocation of CPUs. This basic mechanism allows any node to launch a discovery process, that needs only to follow the information of free CPUs in each branch. Results from experiments and simulation tests, using a simple allocation method, show discovery time scaling logarithmically with the number of nodes.

1. Introduction

Recent works and solutions in Grid Computing show that centralized management and scheduling of computational resources based on complex economic theories produce very interesant results in terms of equitativity and best use. However, it also proves that an approach between Peer-to-Peer (P2P) and Grid Computing is needed, because scalability and fault-tolerance problems rapidly arise when the number of resources managed is incremented. First steps into this kind of solutions where taken by projects like SETI@Home [3] and distributed.net [11], but they still maintain a master-slave model where only one entity in the network generates the workload, and the rest consume it, leading again to centralized management.

In this paper, we propose an architecture where any participant of the network may need idle cycles to complete its tasks. The idea is to bring distributed and grid computing solutions to users with a lower profile than research laboratories. The availability of virtual machines to limit local resource use makes this situation more feasible. The system manages information about free CPUs so that when a node does not have enough computing power to finish its work, it may divide it into n independent tasks and ask the network to discover n more idle machines in a fast and efficient way. Unlike common P2P resource management systems, like data sharing networks, in this project resources are consumible, meaning that they cannot be used by more than one client at the same time. For this reason free nodes availability information must be kept up to date.

The system consists of a peer-to-peer network based on a balanced tree structure that finds the nearest free CPUs to the one that is demanding the execution of a number of tasks. At any time, any node of the network may request the execution of n tasks; this request is routed by neighbour nodes to those available ones that are closer to the originating client with a fast discovery protocol. The tree structure allows applying different constraints to the idle CPU search by using the information about existing free nodes, that is dynamically managed by an availability protocol. These funtionalities are obtained with little state in nodes, and low communication and CPU overhead. A simple allocation policy has been designed and implemented to evaluate the architecture behavior.

This paper takes some steps into a complete distributed computing solution, thus we will impose some restrictions to the environment: We assume that nodes execute batch tasks that do not communicate between them, so we won't be addressing the issues that arise from having dependencies. Also, we will suppose that there is low churn, that is, joins and leavings are not frequent. And finally, we will only consider a weak concept of fairness in the allocation of free nodes.

The rest of the article will be structured as follows: In Sect. 2 we will expose an overview of the system architecture and its behavior. In Sect. 3 the hierarchical overlay topology and its management will be detailed, followed by the protocols that allow the fast and scalable discovery of

free nodes in Sects. 4 and 5. Finally, in Sect. 6 we will show the experimental results and in Sects. 7 and 8 we will explain what other work has been presented concerning distributed computing in peer-to-peer networks and the conclusions of this investigation.

2. Related Work

As it has been pointed out in the introduction, the main approximation until now to a highly scalable distributed computing environment is one entity harnessing the idle cycles of personal computers donated by volunteers, as in SETI@Home project, the BOINC generic framework [2] and distributed.net. Those projects use the traditional client/server paradigm to schedule tasks and return results, what soon leads to scalability problems. For that reason, more elaborated network structures and distributed algorithms have been adopted. One example is Javelin++ [17], which extends the concepts of Javelin [9] replacing the broker that scheduled the tasks with a network of brokers. Recently, more strict peer-to-peer networks have been used to select the nodes which would execute the tasks. BOINC and similar projects adopt an application-driven perspective, in which the existence of an element that is generating all the workload determines the structure of the network and the management algorithms.

Following a more general view, another family of projects, in which this paper is included, have proposed an architecture where every participant can generate the workload, which is better suited for this peer-to-peer philosophy. CompuP2P [13] is one of the first works to use a decentralized peer-to-peer network to manage processor cycles as a shared resource. It arranges all the nodes in a Chord [19] ring and organizes them into 'compute markets', where idle cycles are traded with. However, it presents a scalability problem because it has no mechanism to limit the number of nodes in a market or to balance load between markets. G2-P2P [16] takes an object-oriented approach. It uses Pastry [18] to create a Distributed Hash Table (DHT) where computation objects are stored. Using an uniform hashing function they claim to achieve a good load-balancing property, but there is no other criterion to select the most appropriate free node. In [7] the Pastry DHT is also used, but exploiting its locality awareness to discover near idle nodes. It announces availability with controlled message floodings, what leads to inefficiency as far free nodes are not discovered. Finally, [8] is a discovery mechanism that uses the computational properties of each machine (CPU, RAM, disk space, ...) to form a Chord identifier, what allows searches by description. However, it forces exact queries, so it cannot do more general and massive searches efficiently. In contrast with these approximations, [4] proposes the use of an unstructured overlay network, as it is easier to manage, and it is traversed with random walks. Even though, that is also inefficient because there is no way of knowing if the next node of the walk is free or not.

In some of these works it can be seen that there is an increasing tendency to implement the task execution environment over a virtual machine. In this way, local resources and security are easily managed, as remote tasks are only allowed to access a limited part of the host node. This method allows setting arbitrary limits to resource usage, like CPU, memory, disk or bandwidth. For this reason, more projects are being developed using interpreted languages; for example, [7] uses the Java Virtual Machine and G2-P2P uses the .NET Platform. Another solution is to use OS-level virtual machines, like Xen [5]; this is the solution presented by [20].

For the overlay topology, other authors have proposed the use of a virtual tree on top of a DHT, where each node store only part of a tree index, mainly oriented to range queries. Examples of this are P-Tree [10], P-Grid [1] and VBI-Tree [14]. However, they rely on a uniform distribution of the shared resource; for example, using a uniform hashing function for the DHT. For that reason, BATON [15] uses a balanced binary tree. This type of organization is better suited for a non-uniform resource distribution because the tree gets balanced automatically when the insertions or deletions occur within the same zone. We adopt these ideas but with more than two children per node.

3. System Architecture

The system has a layered architecture that lets us stablish a separation between network topology and resource management. From the lowest level:

- The first layer defines the *connectivity protocol* that maintains the overlay links in the network. It conforms a tree-based network overlay, derived from the B-Tree [6], thus it is a balanced tree where each node can have between m and $2m$ children and the height is always a logarithm of the number of nodes N. The protocol states how nodes join and leave the network, how the tree is kept balanced, and how node failures are dealt with to rebuild the structure.

- The second layer is described by the *availability protocol*, that distributes information relative to the number of free nodes and computing power each time it changes. Every node of the network stores the global state of the branch that hangs below it in the B-Tree structure, and communicates updates to its parent so it can recompute the state of its own branch. This protocol uses a number of techniques that prevent the upper levels of the tree from being flooded with update notifications, while maintaining the information accurate

enough to maximize the network use. Also, the conservative approach of notification updates yields to a more stable behavior of this protocol.

- Finally, the *discovery protocol* uses the information stored in each branch by the availability protocol to route free nodes requests up and down the tree. It tries to find those free nodes that meet a trade-off between proximity to the client and computing power by distributing the requests among the appropriate branches at each level. Therefore the search is performed in a number of network hops that depends only on the height of the tree and, consequently, on the logarithm of the number of nodes of the network.

- Over these three layers we can implement different resource allocation methods. For the simulation tests we have used a simple one, where each free node discovered is automatically allocated to the client.

With this three-layered structure, the system can be easily extended to other types of resources for the discovery protocol, like memory or bandwidth, and other notification policies for the availability protocol.

4. B-Tree Based Topology

The overlay network topology is a hierarchy where every node of the network is mapped to a node of the tree. Other authors have proposed the use of a virtual tree on top of a DHT, where each node store only part of a tree index, mainly oriented to range queries. However, they rely on a uniform distribution of the shared resource; for example, using a uniform hashing function for the DHT. On the other hand, using a balanced tree where each node of the network maintains one node of the tree is better suited for a non-uniform resource distribution because the tree gets balanced automatically when the insertions or deletions occur within the same zone. In our approximation we use a B-Tree [6] variant; it maintains the balance in every join and departure and allows more than two child nodes, thus reducing the tree height.

The main objective of the tree is grouping nearby nodes. In this way, a node can communicate with the nearest ones to itself because they are its siblings or its descendants, and it can reach other regions of the tree by means of its parent. However, the concept of locality usually depends on many variables, so it is actually an aproximation. We have decided to use the simple yet effective way of organizing the nodes in the tree by their phisical address, actually their IP address. Based on the subnet partitioning of the IP address space and the studies on geographic locality of IP addresses, like [12], this method allows a fast and easy decision of

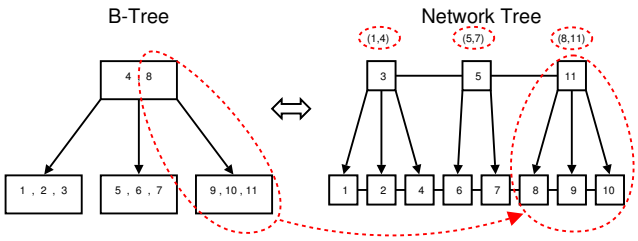

Figure 1. Differences between B-Tree and network tree.

where to insert a node in the tree when it joins the network, while maintaining good metrics (latency and bandwidth, mainly) between nodes of the same branch, specially near the leaves.

Our tree has some differences with the original B-Tree. First of all, as nodes in a B-Tree may hold more than one value (in this case, values are IP addresses), those B-Tree nodes are translated to a group of siblings where each node of the network tree holds only one value; with this one-to-one mapping, every node of the network participates in the management of the tree structure. Also, every inner node (a node with children) has a pair of values that represent the interval of addresses of its descendats and itself. These intervals are used to route messages along the tree, mainly in the operations of insertion and deletion of nodes of the tree. These two main differences can be observed in Figure 1.

Like B-Trees, there exist a constant m so that every node not being the root of the hierarchy always has between m and $2m$ siblings. If these limits are exceeded, then the tree must be rebalanced. When a group of siblings has more than $2m$ nodes, it must be divided into two. On the other hand, if it has less than m nodes it must take some from adjacent groups or be joined with them, eventually. A high value of m priorizes performance for search over network management, because the tree is lower, so this is usually preferred in this context.

Concerning fault-tolerance, every node knows the address of the k predecessors and k successors at the same level (they can be "brothers" or "cousins"). When a node fails, the tree structure can be repaired using these references, because they allow the communication between a node and the brother of its dead father. Obviously, the value of k is an agreement between fault-tolerance and an overload in the management of the tree.

4.1. Joining and Leaving the Network

These are the two main operations that affect the structure of the network; additionally, as a side effect, they can trigger a rebalance. Joining is usually easier: when a node

requests an insertion, the request message is routed through the tree. It goes up the hierarchy looking for a node which interval contains the address of the new node, and then it goes down until it reaches the node with the nearest address to the new node's address. Finally they become brothers and the new node updates its references to its neighbours.

When the father node is notified of the new node, it may request a group split to rebalance the tree if the number of child nodes is greater than $2m$. It stays as the father for one of the new subgroups and asks the other one to designate a leaf node as the new father for that subgroup. This new father node becomes a sibling of the old father, so that their group of siblings is incremented in one member and their respective father must check if it has more than $2m$ child nodes, repeating the process for the next level of the tree. Eventually, a group of siblings is splited in every level until the group of the root is reached, which operates differently.

When a node is added to the group of the root, each of the members checks the number of nodes in that group by counting the references in their successor and predecessor lists. When a root node has k references in both lists (they are full) it looks for a descendant leaf node, which becomes the new root of the tree. This method limits the size of the root to $2k$ nodes, assuming that m is larger than k, but it is very simple because creating a new root node is the decision of only one node; a higher limit would require communication between the members of the group. The high level algorithms can be seen in Fig. 2.

By leaving the network we assume a voluntary action; otherwise, when a node fails the structure must be rebuilt by its neighbours. First of all, a leaving node must check if it has any child. If so it looks for a leaf node that becomes the new father of all of them, similarly to the creation of a new root node. Once done, or if it had no child nodes, it notifies its siblings and its father that it is going to leave and then they update their reference lists.

Similarly to the joining, when the father node is notified of the node leaving, it must check if the number of child nodes is less than m. In that case, it will ask its predecessor or successor to send it child nodes until it has m again. If they both do not sum up more than $2m$ nodes, then they are joined in only one branch. One special case is when the father has no siblings because it is the only one node in the root group. Then it will check if it has less than $2k$ child nodes, in a similar way as it was done in the insertion in the root. If it has so, it will insert itself in the tree, leaving its children as the new root group. All of this can be seen also in Fig. 3.

5. CPU Availability Management

In order to allow the discovery protocol to find free nodes, each inner node must store information about its de-

```
insert(node) {
  if is_my_brother(node) {
    send_references(node);
    if i_have_parent() {
      parent.new_child(node);
    } else
      if num_pred == k and num_succ == k
        look_for_new_root();
  } else route(node);
}

new_child(node) {
  add_child(node);
  if num_children > 2*m {
    new_parent = select_new_parent(children);
    if my_ip < new_parent.ip {
      i = num_children/2; j = num_children;
    } else { i = 0; j = num_children/2; }
    new_parent.set_children(children, i, j);
    num_children = m;
    // Turn new_parent into our brother
    send_references(new_parent);
    if i_have_parent() {
      notify_parent(node);
    } else if num_pred == k and num_succ == k
      look_for_new_root();
  }
```

Figure 2. Insert algorithms

```
delete() {
  if num_children > 0 {
    new_parent = look_for_new_parent();
    i = 0; j = num_children;
    new_parent.set_children(children, i, j);
    num_children = 0;
  }
  notify_siblings();
  parent.child_gone(this);
}

child_gone(node) {
  remove_child(node);
  if num_pred == 0 and num_succ == 0
     and num_children < 2*k {
    children[0].insert(this);
  } else if num_children < m {
    sibling.ask_for_children(num_children);
  }
}

ask_for_children(num) {
  if num_children + num < 2*m
    join();
  else send_children(m - num);
}
```

Figure 3. Delete algorithms

scendants; not exhaustive information, but more general information about the branch as a whole. It knows the approximate number of free CPUs of the branch, the maximum computing power and the minimum number of hops to a free node. This way, the management of this information becomes scalable as it does not depend directly on the number of nodes.

The information each node stores about its branch must be communicated to its parent so that it can efficiently route requests to the idle nodes it has under itself. Therefore, each node not only has information about its branch, but also about each of its sub-branches. The way this information propagates is critical, because it must be kept up to date without flooding the network with notification messages, specially near the root where there are less nodes per level of the tree. This propagation is performed by the availability protocol. Basically, when a node receives a notification of change from one of its child nodes, it must decide if it has to inform its parent, too. With the maximum computing power and the minimum number of hops to a free node, the process is simple. The inner node has to calculate the new maximum and minimum values, respectively, between its child nodes and itself, and if it changes, route the new information to its parent immediately. Note that when a notification goes up one level there is less probability of being the maximum or minimum because the branches are bigger at each hop, so the traffic is self-limited and is unlikely that it reaches the top levels.

The problems arise with the value of the number of free nodes, because when a node gets ready (busy), the number of free nodes of each of its ancestors increases (decreases) by one. If the notification were sent with every change, the root would get informed of all of them, what leads to an unacceptably high traffic in the top levels. For this reason, we have designed a technique that delays the notification of the number of free nodes at each level of the tree, reducing the traffic routed up to the root. The basis of this method lies on sending a notification when the change is meaningful enough. Actually, this means that the most significant bit set to one changes; that is, the number of free nodes crosses a boundary of power of two. For example, a notification would be sent if this value changes from 7 to 8 (111b and 1000b in binary) or from 32 to 31 (100000b and 11111b), but not when it changes from 23 to 24 (10111b and 11000b). Although this yields to a precision lack, there are three main reasons for using this technique:

1. Trying to provide optimality based on exact information is senseless when we are dealing with millions of nodes that are continually and concurrently changing state.

2. The traffic of notifications in the top levels is reduced because as a notification goes up the tree it is less probable of being routed to the next level. This depends also on the number of free nodes, as a high number has also less probabilities of being routed.

3. When the number of free nodes is low, the precision of this value is better. This is most relevant as the nodes of the network get busy, because they are correctly well-spent when there last only a few free nodes.

4. When the number of free nodes changes, this method forces a stabilization mechanism in the propagation of this value.

There are two policies deciding what availability value to take as reference for a branch when a child node notifies a change to its father: optimistic and conservative. An optimistic policy would use the same value sent by the child. On one hand, it has the advantage of having better precision in the information each node stores about its branch, but on the other hand the real number of free nodes of a branch could decrease and be less than the number its father is using, making top level nodes route requests to branches that cannot cope with them. For this reason we have decided to adopt a conservative policy, which would store a lower value, for example the higher power of two less than

```
update(child, free_nodes,
       max_power, min_hops) {
  children[child].free = free_nodes;
  children[child].power = max_power;
  children[child].hops = min_hops;
  // Save old values
  old_power = power;
  old_hops = hops;
  old_free = free;
  // Update value of power, hops and free
  update_values();
  lbo = last_bit_one(free);
  old_lbo = last_bit_one(old_free);
  if power > old_power or
     hops < old_hops or
     lbo != old_lbo
    parent.update(this, free, power, hops);
}
```

Figure 4. Notification algorithm

or equal to the notified value. With such a policy, the system has a better behavior against situations when there last very few free nodes, as it delays too big requests, although it does not make the most of the network. This helps making the protocol stable and provides a gradual convergence in the occupation of the network.

6. Discovery of Free CPUs

As it has been said, when a node has a number of tasks to be done, it requests the network to find that number of idle machines. This service is accomplished with the discovery protocol, that uses the information collected by the availability protocol. By applying heuristic rules, it will try to allocate the fastest and nearest free nodes, so that tasks execution is efficiently done.

A node that receives a message with n pending tasks will first check if itself is ready to execute a new one. If so, it takes one of the tasks from the message. Then it distributes the remaining tasks between its child nodes according to the number of free nodes each branch has, giving priority to the branches having more computing power or less hops to a free node. If it is not enough with the children branches to cope with all the tasks, then a new message with the last tasks will be sent to the father so that it can reach more distant branches. When the message arrives at the root of the tree and it cannot be sent to another branch, it is returned to the originating node meaning that there are no free nodes left in the network. The algorithm for each node can be seen in Fig. 5, where each children node has a `free` field for the number of free nodes of the sub-branch, a `power` field for the maximum power and a `hops` field for the minimum

```
discover(msg) {
  if i_am_ready {
    start_task; msg.num_tasks--;
  }
  while msg.n > 0 or branch_full {
    for i in children {
      if i.free > max.free or
         (i.free == max.free and
          (i.power > max.power or
           (i.power == max.power and
            i.hops < max.hops)))
        max = i;
    }
    add_task_to(max);
    msg.num_tasks--;
  }
  if msg.num_tasks > 0 {
    if i_have_father
      send_father(msg);
    else
      send_client_node(msg);
  }
}
```

Figure 5. Discover protocol algorithm

number of hops to a free node.

The worst case would be that of a leaf node that needs to allocate every node of the net. The request would have to go up to the root and then down to the rest of the tree; that is the longest path a request would traverse. As discovery of idle nodes is done concurrently in every branch, that would be the same as reaching one idle leaf node in the opposite side of the tree, which is done in $O(2\log_m N)$ hops, being N the number of nodes in the network, thus making the discovery protocol highly scalable.

This is a best effort network. That is, the intermediate nodes make its best to route the message to the most suitable free nodes, but the reception is not guaranteed. In fact, the failure of nodes is frequent in a peer-to-peer network. For this reason, both the originating node and the allocated ones must avoid problems in the discovery phase and when sending the actual work to execute using timeout mechanisms, acknowledge messages and retransmissions.

7. Experimental Results

This architecture has been validated with a simulator, implemented with the OMNeT++ simulation framework. The allocation policy used in the tests has been a simple one, where as soon as nodes are discovered they are allocated, because we are focused on resource discovery performance.

Tests have been done aimed to measure free nodes discovery time, control messages traffic and CPU load. Every

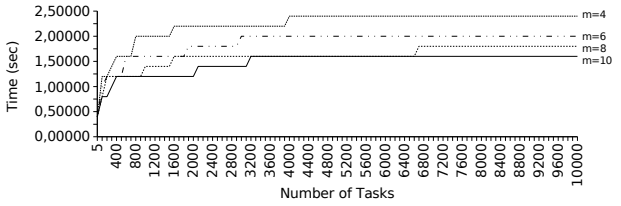

Figure 6. Discovery time for as many free nodes as requested tasks. The test network has 50000 nodes and one hop is 200 ms.

test has been issued with variations in the number of nodes, N, and the B-Tree parameter, m, to study the impact of the size and structure of the network in the performance of the protocols. The simulations have been performed with up to 50000 nodes and values of m from 6 to 10. Variations on the duration of the tasks and the size of the data have also been applied to recreate more realistic situations. There are three constants, though: the latency of the network connections has been established to 200 ms, the mean continental value for Internet, to simulate a very wide area network; 1 Mbps has been taken for the bandwidth, a conservative value for a home Internet connection; and the mean computing power of the nodes has been set to 2000 MIPS.

Time tests show that both the number of nodes and the number of child nodes per parent affect the discovery of free nodes. Just as expected, the last free node of the n requested is reached in $O(2\log_m n)$ hops. For this reason, a network with a higher value of m performs better, while an increasing value of n is hardly appreciated. The results of the free nodes discovery time tests can be seen in Fig. 6 as a logarithmic growth. We can extrapolate the results to higher values of n. For example, we calculate that, for the test network, requesting the execution of 100,000 tasks would discover 100,000 free nodes in 2 seconds, 1,000,000 in 2.4 seconds, 10,000,000 in 2.8 seconds, and so on.

Control traffic (traffic of non-data messages) and CPU load tests have been done under two situations: participants have a normal and high activity. Normal activity means that there are frequent requests from randomly chosen nodes, but the network does not get completely busy. Under high activity, every node is busy and continuously receiving new requests. Traffic has been measured in bytes per second. CPU load is more difficult to measure in a simulation, but as every message is managed in nearly constant time (some hundreds of instructions) we have decided to express it in terms of messages per second. Tables 1 and 2 show the results of the normal and high-activity behavior. They present the value of m, the tree height and the mean and maximum values of CPU load and control traffic for the root and leaves of a network of 50000 nodes.

Table 1. CPU load and control traffic under normal activity. The net has 50000 nodes, with $m = 10$ and a bandwidth of 1Mbps.

Tree		Load (msg/s)		Traffic (Bps)	
m	height	mean	max	mean	max
Root					
4	7	0.08	2.87	24.10	519.98
6	6	0.09	5.64	24.98	1005.69
8	5	0.09	5.64	25.28	1005.35
10	5	0.09	5.62	25.43	999.48
Leaves					
4	7	3.56	4.21	1235.89	1343.51
6	6	3.71	4.33	1293.18	1405.28
8	5	3.85	4.56	1330.41	1436.15
10	5	4.12	5.77	1435.08	1485.60

Table 2. CPU load and control traffic under high activity. The net has also 50000 nodes, with $m = 10$ and a bandwidth of 1Mbps.

Tree		Load (msg/s)		Traffic (Bps)	
m	height	mean	max	mean	max
Root					
4	7	0.13	28.20	38.19	4980.56
6	6	0.13	28.46	39.53	5021.05
8	5	0.14	27.65	39.75	4526.82
10	5	0.14	28.44	40.05	5018.49
Leaves					
4	7	39.68	41.62	14359.74	16031.69
6	6	44.20	50.55	15198.24	16205.83
8	5	54.15	59.85	17417.23	19256.67
10	5	63.72	65.29	19566.90	21947.28

In these tables it can be seen that while the discovery protocol was positively affected by the value of m, the overall system load suffers when the tree is lower, thus a trade-off is needed between them. Besides this, by using the availability protocol, under normal behavior both control traffic and CPU load is heavier at the leaves than at the root nodes. Also, control traffic hardly reaches 1KBps, what represents less than 1% of the total bandwidth. However, under high activity rate the root suffers waves of very high CPU load and control traffic.

Results are promising. As we can see in Tables 1 and 2, control overhead is very low. Under normal activity, the control traffic is only 1485 Bps and the CPU load only reaches 5.77 messages per second, in the worst case. And under heavy activity, which would be a very uncommon situation, the control traffic is 21947 Bps and the CPU load is 65.29 messages per second.

8. Conclusions and Future Work

In this paper, we have presented a network architecture that discovers the presence of idle machines with a scalable ($O(log_m N)$) and fast (1.8 seconds for 10000 requested tasks) method. It organizes the nodes in a balanced tree structure to efficiently distribute information about free nodes in a per-branch basis, that is eventually used to route the request from a client to the appropriate idle CPUs. The connectivity protocol, discovery protocol and availability protocol are all three designed in a totally distributed way, that provide high scalability and fault-tolerance. Moreover, the experimental simulation results show low overhead in the control traffic and CPU load.

We envision to validate these results with a real prototype to be implemented over the PlanetLab testbed. Also, a distributed simulator in development could validate the model in the range of millions of nodes. We believe this can be a valuable step to develop system support for high performance computing applications.

References

[1] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 179–194, London, UK, 2001. Springer-Verlag.

[2] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID*, pages 4–10, 2004.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[4] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama. Unstructured peer-to-peer networks for sharing processor cycles. *Journal Parallel Computing (PARCO)*, 32(2):115–135, February 2006.

[5] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[6] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*, pages 107–141. ACM, 1970.

[7] A. R. Butt, X. Fang, Y. C. Hu, and S. P. Midkiff. Java, peer-to-peer, and accountability: Building blocks for distributed cycle sharing. In *Virtual Machine Research and Technology Symposium*, pages 163–176, 2004.

[8] A. S. Cheema, M. Muhammad, and I. Gupta. Peer-to-peer discovery of computational resources for grid applications. In *Proc. IEEE/ACM Workshop on Grid Computing (GRID)*, 2005.

[9] B. O. Christiansen, P. R. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-based parallel computing using java. *Concurrency - Practice and Experience*, 9(11):1139–1160, 1997.

[10] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 25–30, New York, NY, USA, 2004. ACM Press.

[11] Distributed.net. http://www.distributed.net, 2000.

[12] M. Freedman, M. Vutukuru, N. Feamster, and H. Balakrishnan. Geographic Locality of IP Prefixes. In *Internet Measurement Conference (IMC) 2005*, Berkeley, CA, October 2005.

[13] R. Gupta and A. K. Somani. Compup2p: An architecture for sharing of computing resources in peer-to-peer networks with selfish nodes. In *Online Proceedings of Second Workshop on the Economics of Peer-to-Peer Systems*. Harvard University, June 2004.

[14] H. V. Jagadish, B. Ooi, Q. Vu, R. Zhang, and A. Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *22nd IEEE International Conference on Data Engineering (ICDE), 2006 (to appear)*, 2006.

[15] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.

[16] R. Mason and W. Kelly. G2-p2p: A fully decentralised fault-tolerant cycle-stealing framework. In *ACSW Frontiers*, pages 33–39, 2005.

[17] M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: scalability issues in global computing. *Concurrency: Practice and Experience*, 12(8):727–753, 2000.

[18] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[19] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[20] A. I. Sundararaj and P. A. Dinda. Towards virtual networks for virtual machine grid computing. In *Virtual Machine Research and Technology Symposium*, pages 177–190, 2004.