Práctica 5.ª
Introducción a la programación distribuida

Escuela de Ingeniería y Arquitectura
Depto. de Informática e Ingeniería de Sistemas

1. Objetivos

1. Programar de forma distribuida en Java o Ada con *sockets*
2. Introducir el concepto de *computación en malla* o *grid computing*

2. Introducción

En esta práctica se va a simular un sistema de computación en malla (en inglés, *grid computing*) utilizando una arquitectura cliente-servidor basada en la utilización de canales de comunicación síncronos (en concreto, *sockets*). Esta práctica puede ser programada tanto en Ada como en Java.

El concepto de *computación en malla* hace referencia a la combinación de recursos informáticos heterogéneos, dispersos geográficamente y débilmente acoplados para alcanzar un objetivo común. La malla puede verse como un sistema distribuido en el que cada nodo ejecuta parte de una tarea conducente a satisfacer ese objetivo común que requiere, por lo general, un elevado tiempo de computación y poca comunicación con otros nodos del sistema. Los nodos se comunican utilizando una red de comunicaciones (generalmente, Internet).

SETI@home[1] es un proyecto de computación de la Universidad de California en Berkeley. SETI es un acrónimo de *Search for Extra-Terrestrial Intelligence* y su propósito

[1] `http://setiathome.ssl.berkeley.edu/`

es el de analizar las señales de radio recibidas por el radiotelescopio de Arecibo (Puerto Rico) para buscar en ellas una prueba de la existencia de vida inteligente extra-terrestre.

Lanzado al público en 1999, SETI@home se basa en que los análisis de las señales recibidas (costosos en tiempo) son realizados utilizando el tiempo de CPU sobrante de los computadores que, de forma voluntaria y gratuita, son puestos a disposición del proyecto por sus propietarios. Puede considerarse como uno de los primeros intentos de computación distribuida en malla realizados con éxito y en el que participan voluntarios de todo el mundo. Desde 2005, se ejecuta a través de la plataforma para computación distribuida BOINC.

Los computadores de los voluntarios solicitan a través de Internet a un servidor del proyecto SETI una o varias *unidades de trabajo* (fragmentos de 107,4 s de señal grabada en el radiotelescopio de Arecibo), las analizan y devuelven los resultados al servidor. Este se encarga de la gestión del envío de dichas unidades de trabajo y de la recepción de los resultados. Cada unidad de trabajo se manda a más de un cliente, con el objetivo de poder detectar si algún cliente se comporta de forma anómala o falsea los resultados.

3. Descripción del trabajo a realizar

En esta práctica se va a simular el funcionamiento de SETI@home utilizando tareas de Ada o hilos de ejecución de Java, y realizando la comunicación por internet a través de *sockets*. Se suministra un programa ya implementado que simula el funcionamiento del servidor SETI@home y debe implementarse en Ada o Java un cliente que interactúe con dicho servidor.

El programa servidor se encuentra disponible en el clúster *hendrix*, en el directorio de prácticas de la asignatura[2]. El fichero «`servidor_seti`» es la versión para ser ejecutada en *hendrix* y «`servidor_seti.exe`», la versión para Windows. El servidor admite, en el momento de ser lanzado desde la línea de comandos, un argumento que especifica el número de puerto en el que va a quedar a la escucha de peticiones. Si no se especifica dicho argumento, utiliza por defecto el puerto 5432. Para evitar colisiones cuando lo ejecutéis, se recomienda que utilicéis vuestro NIP como número de puerto.

El servidor dispone de un total de `NUM_CLIENTES` unidades de trabajo que deben ser procesadas por los clientes. En nuestro contexto, los datos asociados a cada unidad de trabajo van a ser simplemente números naturales aleatorios pertenecientes al dominio de valores de los tipos `integer` de Ada o `int` de Java. El procesamiento de los datos asociados a cada unidad de trabajo, que deberán llevar a cabo los clientes, será simplemente el cálculo de la suma de los dígitos de ese número.

Se debe construir un programa cliente en el que se ejecuten de forma concurrente un

[2] «`hendrix-ssh:/export/home/practicas/Practicas/concurrente`» visto desde Unix, «`\\hendrix-cifs\Practicas\concurrente`» desde Windows

total de `NUM_CLIENTES` procesos clientes de SETI@home. Cada proceso cliente, de forma iterativa, solicitará al servidor el envío de una unidad de trabajo, la procesará (calculando la suma de sus dígitos), simulará que está realizando un cálculo más complejo mediante una espera aleatoria de entre 0 y `MAX_TIEMPO_TRABAJO` segundos y devolverá el resultado al servidor. Todos los clientes informarán, escribiendo en la pantalla, cuando reciban una unidad de trabajo y cuando la devuelvan.

Uno de los clientes deberá tener un comportamiento anómalo o malicioso y enviará resultados incorrectos para todas las unidades de trabajo que se le asignen.

Para la implementación, pueden utilizarse los siguientes valores para inicializar las constantes referidas anteriormente:

Constante	Valor
NUM_CLIENTES	20
NUM_UNIDADES_TRABAJO	20 000
MAX_TIEMPO_TRABAJO	2 s

4. Descripción del protocolo de comunicaciones con el servidor

El servidor cuyo ejecutable se facilitan sigue el siguiente protocolo para comunicarse con los clientes:

- Espera en el puerto que se ha especificado al lanzarse (o en el 5432 en el caso de que no se haya especificado ninguno) a que se conecte algún cliente.
- Cuando un cliente se conecta a su puerto de escucha, espera recibir los siguientes datos en este orden:
  - Un *byte* cuyo valor representa el número que identifica al cliente.
  - Un *byte* que codifica el tipo de operación que el cliente va a realizar con el servidor: 0 si quiere solicitar una nueva unidad de trabajo; 1 si quiere enviar un resultado.
  - Solo en el caso de que el tipo de operación sea el envío de un resultado: 4 *bytes* indicando el identificador de la unidad de trabajo cuyo resultado se entrega. El primero de los *bytes* transmitidos es el *byte* **más significativo** del identificador, y el cuarto, el **menos significativo**.
  - Solo en el caso de que el tipo de operación sea el envío de un resultado: un *byte* indicando el resultado del procesamiento de la unidad de trabajo cuyo resultado se entrega.

- Cuando un cliente ha solicitado un trabajo, el servidor responde con la siguiente información utilizando la misma conexión (el mismo *socket*) con el que se hizo la petición:

  - 4 *bytes* indicando el identificador de la unidad de trabajo cuyo resultado se entrega. El primero de los *bytes* transmitidos es el *byte* **más significativo** del identificador, y el cuarto, el **menos significativo**.
  - 4 *bytes* que representan los datos asociados a la unidad de trabajo. El primero de los *bytes* transmitidos es el *byte* **más significativo**, y el cuarto, el **menos significativo**.

  Si no puede encargar al cliente ningún trabajo, porque ya todos están procesados o asignados, envía al cliente 8 *bytes* iguales a cero. El cliente, cuando lo recibe, debe dejar de solicitar trabajos al servidor y finaliza.

5. *Sockets* en Ada

En Ada, puede utilizarse el paquete `GNAT.Sockets` para trabajar con *sockets*. En la dirección `http://www.radford.edu/~nokie/classes/320/std_lib_html/gnat-sockets.html` puede encontrarse documentación de dicho paquete y en el directorio de prácticas de la asignatura en `hendrix` hay un ejemplo, denominado `PingPong`, extraído de la especificación del paquete `GNAT.Sockets` que muestra un ejemplo de utilización de los sockets en un único programa con dos tareas: `Ping` (cliente) y `Pong` (servidor).

En él se pueden ver las acciones que es necesario realizar por parte de un servidor para crear y utilizar un *socket* (`Server` en el ejemplo):

- Creación del *socket* con el procedimiento `Create_Socket`.
- Configuración con el procedimiento `Set_Socket_Option`.
- Asociación a un puerto con el procedimiento `Bind_Socket`.
- Especificación de que el *socket* es de escucha y asociación de una lista de conexiones de entrada (`Listen_Socket`).
- Aceptación de una solicitud de conexión y creación de un nuevo *socket* para gestionar dicha conexión con el procedimiento `Accept_Socket`.
- Obtención del *stream* asociado al *socket* de la conexión para enviar o recibir información con el cliente a través de la función `Stream`.
- Cierre del *socket* asociado a la conexión (`Close_socket`).
- Una vez que el proceso servidor termina, liberación de los recursos asociados al *socket* (`Close_Socket`).

En el caso del cliente, las acciones necesarias son:

- Creación del *socket* con el procedimiento `Create_Socket`.
- Configuración con el procedimiento `Set_Socket_Option`.
- Conexión con el servidor utilizando el procedimiento `Connect_Socket`.
- Obtención del *stream* asociado al *socket* de la conexión para enviar o recibir información con el cliente a través de la función `Stream`.
- Cierre del *socket* asociado a la conexión (`Close_socket`).

Antes de utilizar los *sockets* del paquete `GNAT.Sockets`, hay que ejecutar el método `Initialize`, y al terminar, el método `Finalize`.

La comunicación a través de *sockets* se realiza a través de datos de tipo *stream*, similares a ficheros secuenciales de datos de tipo heterogéneo y cuya definición se encuentra en la sección 13.13 del manual de referencia de Ada.

Para realizar la práctica en Ada, puede ser útil definir el tipo de datos entero *byte* de la siguiente forma: `type byte is mod 256;`

6. *Sockets* en Java

En Java, pueden utilizarse las clases `Socket` y `ServerSocket` del paquete `java.net`. El API de Java[3] y el tutorial *All About Sockets*[4] proporcionan información útil para utilizar *sockets* en Java.

En Java, un servidor que tenga que crear un *socket* para la escucha de peticiones en un determinado puerto, debe utilizar un objeto de la clase `ServerSocket`

- Debe crearlo, asociándolo con un puerto en el momento de la construcción.
- Debe permanecer a la escucha de peticiones de clientes con el método `accept()`. Este método, cuando se produce una nueva petición de un cliente, devuelve un nuevo objeto de la clase `Socket` para gestionar dicha conexión.
- Debe obtener los objetos `InputStream` y `OutputStream` asociados al objeto de la clase `Socket` para comunicarse con el cliente.
- Cuando ha terminado la comunicación con ese cliente en particular, se deben liberar los recursos del objeto de la clase `Socket` invocando al método `close()`.

[3] `http://docs.oracle.com/javase/7/docs/api/`
[4] `http://docs.oracle.com/javase/tutorial/networking/sockets/`

- Una vez que el proceso servidor termina, debe liberar los recursos asociados al *socket* de la clase `ServerSocket` invocando el método `close()`.

Las acciones ejecutadas por un cliente para conectarse a un servidor son las siguientes:

- Creación de un objeto de la clase `Socket` especificando la dirección y puerto del servidor.
- Obtención de objetos `InputStream` y `OutputStream` para la comunicación con el servidor.
- Cierre del *socket* asociado a la conexión (`Close_socket`).

La comunicación en el caso de Java, puede hacerse a través de de objetos `InputStream` (para la lectura de datos) y `OutputStream` (para la escritura de datos). El tutorial *I/O Streams*[5] proporciona información sobre el uso de *streams* en Java.

7. Entrega de material

Como resultado de esta práctica hay que enviar por correo electrónico al profesor de prácticas (`latre@unizar.es`) un fichero comprimido en formato ZIP denominado «`pract5.zip`» que contenga todo el código fuente necesario para compilar y ejecutar el programa solicitado.

La fecha límite de entrega es el 5 de junio de 2012.

[5] `http://docs.oracle.com/javase/tutorial/essential/io/streams.html`