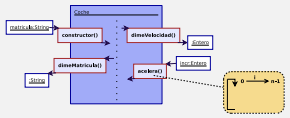
Grado en Ingeniería de Sistemas de Telecomunicación, Sonido e Imagen

Programación 2



Práctica 2

UNIVERSIDAD POLITECNICA DE VALENCIA

Escola Politècnica Superior de Gandia

DSIC

Departament de Sistemes Informàtics i Computació

Contenido

Sesiones

- Grupos lunes: 11-mar, 25-mar, 15-abr, 22-abr
- Grupos viernes: 15-mar, 22-mar, 12-abr, 19-abr
- Examen: 29-abr (10-12h)

Objetivos

- Aprendizaje de clases de biblioteca (ficheros, threads, sockets) mediante la lectura de ejemplos.
- Introducción a la computación distribuida. El modelo cliente-servidor. Factores a considerar:
  - comunicación y sincronización entre cliente y servidor: el concepto de protocolo.
  - Tipo de atención: secuencial o paralela. Sincronización interna del servidor.
- Lectura del código típico de un programa cliente-servidor.
- Ingeniería inversa del código de una clase.
- Implementación de métodos dados su diseño y su algoritmo.
- Implementación de un protocolo sencillo de transmisión de ficheros, similar al HTTP.

¡ Atención !

▷ Se recuerda que las prácticas deben prepararse antes de acudir al aula informática. Esto incluye leer el código proporcionado y probarlo.

▷ La realización de las prácticas es un trabajo individual y original. En caso de plagio se excluirá al alumno de la asignatura. Por tanto es preferible presentar el trabajo realizado por uno mismo aunque éste tenga errores.

1 Introducción a la computación distribuida

El impacto que la transmisión de información (voz, imagen, texto) ha tenido en el desarrollo mundial, social y económico, es tan grande que resulta imposible de cuantificar.

Pero este efecto, realmente se disparó al pasar de una comunicación básicamente en sólo un sentido (televisión y radio, exceptuando el teléfono) a una comunicación en los dos sentidos, disponible de forma popular y generalizada.

Esta comunicación bidireccional generalizada ha sido posible gracias a los ordenadores que, mediante programas, han permitido el procesamiento y transmisión de información, una vez representada ésta de forma numérica[1] (codificación[2]).

Por ejemplo, gracias a los programas informáticos, se puede transmitir de forma fiable por un canal de transmisión inicialmente no fiable, se puede gestionar el tránsito de información por una red, encaminando cada unidad de datos de forma correcta para que llegue a su destino; y, por su puesto, se pueden desarrollar aplicaciones para que el usuario final pueda compartir o acceder a la información, y pueda en definitiva comunicarse con otras personas.

Todas las aplicaciones de ordenador que hacen posible la comunicación, bien sean programas "de infraestructura" (gestión básica de la comunicación) o bien sean programas "para el usuario final" comparten una característica definitoria. Son aplicaciones distribuidas. Una aplicación distribuida está formada por, al menos, dos procesos[3] ejecutándose en máquinas distintas que intercambian información para conseguir un *fin común*[4].

La relación entre los procesos que conforman una aplicación distribuida son en general complejos, puesto que hay varios factores que deben ser considerados:

---

[1] Digital.
[2] Codificar es representar símbolos mediante números.
[3] Un proceso es un programa en ejecución.
[4] "Fin común" que, por su puesto, o bien necesita o bien es el propio intercambio de información.

- Qué herramientas pueden utilizar dichos procesos para comunicarse y qué garantías ofrece la comunicación. En este contexto se sobrentiende que la comunicación se realiza mediante el modelo de *envío de mensajes*.[5]
- El turno de palabra. Esto es, el orden que determina, en cada momento, cuál es proceso al que corresponde enviar información (y consecuentemente qué proceso debe recibirla).
- El formato que deben seguir los mensajes intercambiados.
- Qué información de control[6] han de intercambiar los procesos para garantizar el éxito de la transmisión, o en su caso, detectar e informar sobre posibles errores.

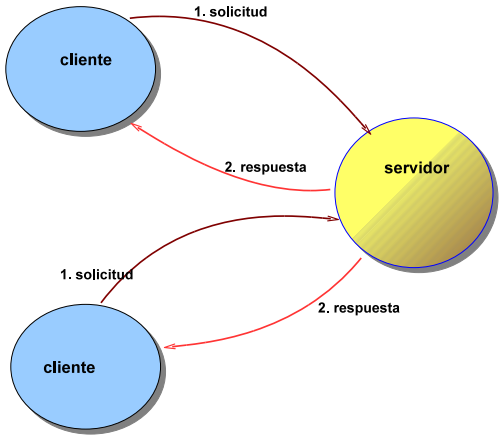
Protocolo

Los tres últimos puntos de la lista anterio se corresponden con la noción de protocolo. Un protocolo es la definición formal de los mensajes digitales y del conjunto de reglas utilizadas para su intercambio entre dos, o más, procesos. En definitiva, un protocolo establece cómo se realiza el diálogo entre los procesos de una aplicación distribuida particular.

Para reducir la complejidad en el diseño de aplicaciones distribuidas resulta conveniente utilizar algún modelo que simplique esta tarea. El más ampliamente usado es el modelo cliente-servidor, no sólo por su sencillez sino porque en la práctica, muchos problemas (aunque no todos) casan bien con este planteamiento.

[5] Y no mediante el modelo de *memoria compartida* ya que, por estar en máquinas distintas no hay una memoria física común a los procesos.

[6] Distinta de la que se prentende transmitir originalmente.

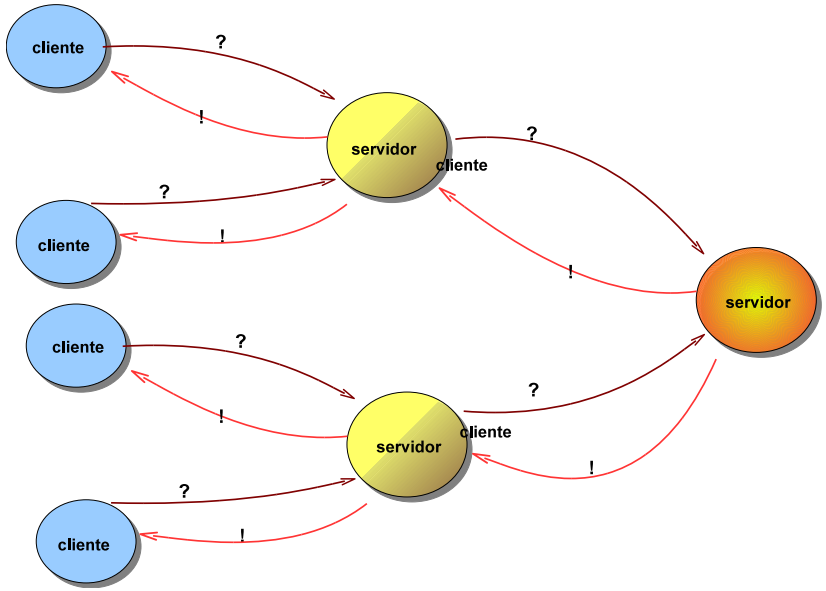


Cliente - Servidor

El modelo cliente-servidor clasifica los procesos de una aplicación distribuida en dos tipos: clientes y servidores. En la versión más sencilla se tiene sólo un proceso servidor y varios clientes (aunque podría ser sólo un cliente). En cualquier caso, el diálogo que se sigue en este modelo consiste en que el cliente realiza una petición, enviando un mensaje al servidor, y este último responde enviando un mensaje al cliente[7]. Por su puesto, en la práctica cliente y servidor podrán intercambiar más de un mensaje, pero siempre será el cliente quien inicie el diálogo y el servidor quien responda. Así pues, podemos deducir que el servidor debe estar siempre disponible, empezando a ejecutarse antes que los clientes y esperando mensajes de ellos. Por contra, un proceso cliente puede ser efímero, iniciarse en cualquier momento y terminar tan pronto como reciba la respuesta del servidor.

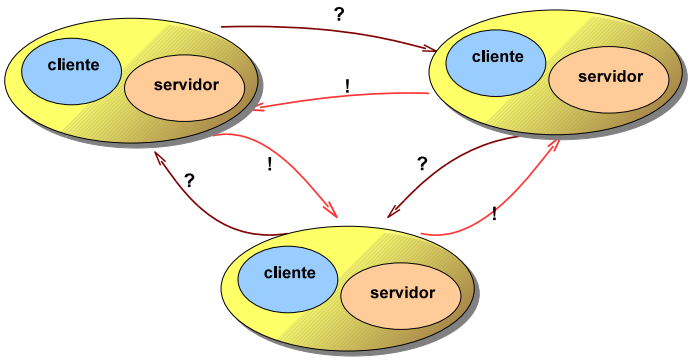
Hemos ya indicado que el modelo cliente-servidor soluciona un buen número de problemas prácticos de diseño de aplicaciones distribuidas, pero no todos.

[7] Esto se conoce también como comunicación *pull*.



Se pueden encontrar problemas en los que se requiera que un servidor envíe información a un cliente sin que éste la hubiera solicitado previamente[8]. También hay ocasiones en que conviene que un servidor sea cliente de otro. Es decir, habría una especie de super-servidor que tendría como clientes a un conjunto de procesos que, ordinariamente son servidores de clientes finales.

[8] Comunicación *push*.



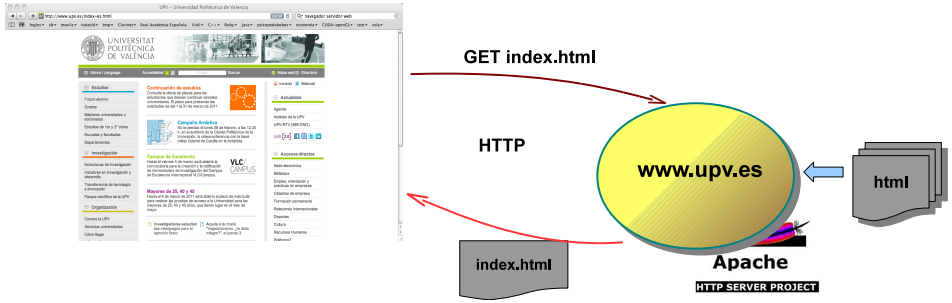
Peer-to-Peer

Y en general, puede resultar necesario que cualquiera de los procesos de una aplicación distribuida tenga la capacidad de iniciar un diálogo o la obligación de responder cuando se le requiere. Cuando esto último ocurre estamos ante el modelo peer-to-peer (P2P) que literalmente quiere decir entre pares[9] y que se solventa en la práctica haciendo que cada proceso se comporte al mismo tiempo como cliente y servidor. Es fácil imaginar que los protocolos en un sistema P2P son más complejos que en el caso cliente-servidor.

[9] Entre iguales, nadie es más que nadie.

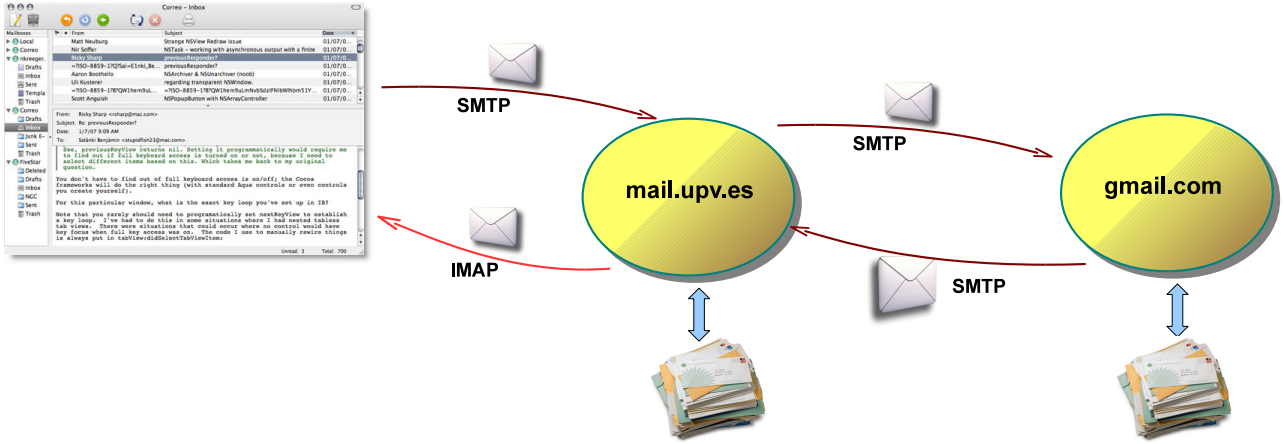
1.1 Ejemplos de aplicaciones distribuidas

- World Wide Web. Se trata del archiconocido sistema de distribución de información basado en páginas enlazadas entre sí, escritas en html (*Hipertext markup language*).
  Esta aplicación es puramente cliente-servidor. Un proceso servidor (el *servidor web*), que siempre está en ejecución, devuelve páginas web almacenadas el disco de su máquina cuando un cliente lo solicita. Por su parte, los clientes, los *navegadores web*, se conectan a un servidor para pedir una de las páginas html que éste guarda. Cuando el navegador recibe la página html, la representa gráficamente. La página html contiene enlaces a otras páginas que pueden estar en el mismo o en otro servidor web. Cuando se pulsa uno de esos enlaces, el navegador sabe por el enlace a qué servidor debe pedir página correspondiente a dicho enlace.
  En resumen, en esta aplicación distribuida encontramos múltiples servidores (servidores web) y múltiples clientes (navegadores) con la particularidad que los clientes no se relacionan siempre con el mismo servidor.



HTTP El protocolo que rige la interación entre clientes y servidores del world wide web es el HTTP (*Hipertext Transfer Protocol*).

- e-mail. En el caso del correo electrónico nos encontramos ante una relación cliente-servidor, aunque con matices. Un servidor de correo realiza dos tareas. Almacena los correos que llegan con destino a los usuarios del servicio (correo entrante), y también se encarga de recibir y distribuir (correo saliente) los correos que los usuarios escriben a otros destinatarios.
  El cliente de correo (o lector de correo) de un usuario se conecta siguiendo el modelo cliente-servidor a su servidor de correo, de donde puede descargar los mensajes destinados al usuario y a quien transmite para su distribución los correos escritos por dicho usuario.
  El importante matiz a tener en cuenta en caso del e-mail[10], es que los servidores de correo no interactúan sólo con clientes lectores de correo, sino también entre sí para intercambiar los mensajes que posee un servidor pero cuyo usuario destino está antendido por otro. Por tanto, la relación entre los servidores de correo es peer-to-peer porque que todo servidor de e-mail puede comportarse como cliente de otro para enviarle correos o como servidor éste cuando tiene que recibirlos.



SMTP, IMAP

El protocolo que se ejecuta para enviar correos electrónicos es el SMTP (*Simple Mail Transfer Protocol*). Para recibir correos un lector puede utilizar el protocolo POP (*Post Office Protocol*) o el IMAP (*Internet Message Access Protocol*).

[10] Que no ocurre en el world wide web.

- Sistemas de intercambio de ficheros. Se trata de una aplicación distribuida que permite que un grupo de usuarios compartan ficheros sin que exista un servidor que medie los intercambios. Popularmente son conocidos como redes p2p, pero el nombre es parcialmente incorrecto. Son efectivamente sistemas peer-to-peer, puesto que no se basan en el modelo cliente servidor, pero no son los únicos sistemas peer-to-peer. Las virtudes de un sistema de intercambio p2p son la tolerancia a fallos y la velocidad de descarga. Al no haber un servidor central, los ficheros compartidos se encuentran repartidos, y de hecho particionados y puede que replicados, entre los distintos procesos de la aplicación. Por eso, el fallo en uno de los procesos no compromete la disponibilidad del servicio. Además, las descargas son más rápidas porque se realizan desde las diferentes máquinas en las que un fichero está repartido, y no desde un único servidor que se saturaría fácilmente con muchos clientes y ficheros grandes.

1.2 Infraestructura de programación para el desarrollo de aplicaciones distribuidas

Para programar una aplicación distribuida necesitamos

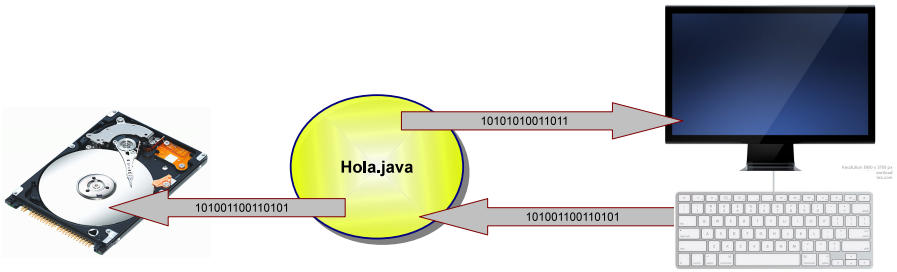
- evidentemente, una forma de comunicación entre los procesos que componen la aplicación distribuida.
- para casos concretos, leer un fichero que deba enviarse y, a su recepción, crear un fichero en donde guardarlo.
- para algunos procesos servidores con gran demanda, conseguir que puedan atender a múltiples clientes simultáneamente (en paralelo).

Para cualquiera de los tres objetivos anteriores se precisa la *colaboración* del sistema operativo, ninguno puede conseguirse únicamente programando desde cero. Esta colaboración la obtendremos mediante clases y métodos de biblioteca.

1.2.1 Flujos

Hay una abstracción que juega un papel importante en la transmisión de datos, no sólo entre procesos distribuidos, sino también entre un proceso y su *entorno*. Se trata de los flujos (stream en inglés). Un flujo es literalmente una corriente (un rio) que lleva datos (bytes, caracteres) de un lugar a otro:

- cuando escribimos en pantalla hay un flujo de caracteres de salida de nuestro programa al monitor
- cuando leemos de teclado hay un flujo de entrada de caracteres del teclado a nuestro programa
- cuando escribimos en un fichero hay un flujo de caracteres de salida de nuestro programa al fichero (y en concreto al disco donde éste se almacena)



- cuando leemos de un fichero hay un flujo de caracteres de entrada del fichero a nuestro programa
- cuando enviamos datos a un proceso en otra máquina hay un flujo de salida de nuestro programa a ese proceso remoto
- cuando nuestro programa recibe datos de un proceso remoto hay un flujo de entrada que viene desde él.

Hemos puesto de manifiesto las similitudes que la idea de flujo consigue al ser aplicada a casos aparentemente diferentes (pantalla, teclado, fichero, conexión de red) ya que con ello vamos a conseguir simplificar la programación. Gracias a la abstracción flujo, enviar información por una red o escribir en un fichero se hará exactamente igual que escribir un texto en pantalla.

Para utilizar un flujo hay tres pasos que siempre deben darse

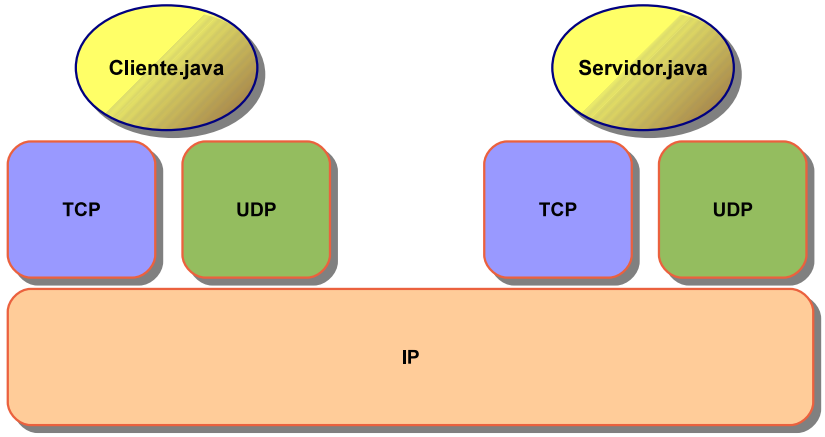
- Apertura del flujo. En este paso se contacta con el recurso externo al programa (fichero, conexión), con la mediación del sistema operativo, y en caso positivo nuestro programa obtiene un objeto que representa el recurso. En la apertura también se indica si el flujo se utilizará como salida (escribir) o como entrada (leer).
- Uso del flujo. Mediante el objeto obtenido en la apertura escribiremos o leeremos, según sea el caso. El efecto de la operación dependerá del recurso al que el flujo está vinculado. Si escribimos en un flujo asociado a una conexión, en realidad estaremos enviando información por la red. Si escribimos en un flujo asociado a un fichero, estaremos guardando lo escrito en el disco.
- Cierre del flujo. En este paso indicamos que nuestro programa ya no necesita trabajar con el flujo, de modo que los cambios pendientes de aplicar al recurso externo se realizan y el sistema operativo libera el espacio que dedicó para su gestión.

1.2.2 Sockets

En el caso concreto de las conexiones de red entre procesos distribuidos existe la abstración llamada socket (zócalo o enchufe) que representa un punto tanto emisión como de recepción de información. Mediante un socket nuestro programa podrá comunicarse con otros. El software que actualmente da soporte a la idea de socket y que permite la comunicación entre redes de ordenadores se conocece con el nombre de Internet Protocol (IP), de donde procede el nombre de internet. En IP aparecen varios conceptos fundamentales además del de socket, como la dirección IP, que es una secuencia de números que identifica a un ordenador conectado a internet. Por ejemplo, 158.42.145.35 significa red 152.42, ordenador en ella 145.35.

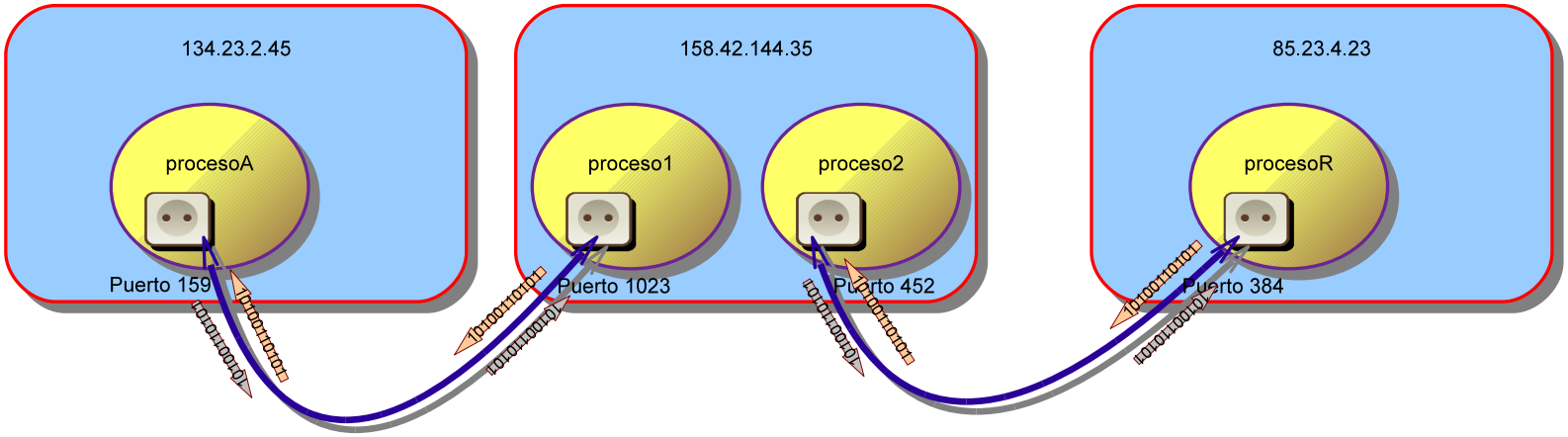
Dirección IP. Puerto

Sobre IP descansan otros dos bloques de software que son los que podemos utilizar para la comunicación ya desde un programa escrito por nosotros. Se trata del Transmission Control Protocol (TCP) y del User Datagram Protocol (UDP). Ambos utilizan el concepto de número de puerto con el mismo fin que los apartado de correos postales. Con un puerto se puede distinguir, dentro de un mismo ordenador, a qué proceso se envía la información (objetivo que no puede lograrse sólo con la dirección IP, que identifica al ordenador en su conjunto).



TCP resulta más sencillo de utilizar que UDP puesto que es fiable (no se pierde información y ésta se recibe en el orden en que se envió) y es orientado a la conexión.

Así pues, cuando utilicemos la combinación de protocolos TCP/IP para comunicar dos procesos tendremos un socket en cada proceso conectado con su homólogo en el otro proceso. Cada socket estará identificado por su dirección IP y su número de puerto. Además un socket TCP tiene vinculados dos flujos uno de entrada para recibir datos y otro de salida para enviarlos. A través de estos flujos podremos enviar texto llano (codificado según la tabla ASCII o similar) o simplemente bytes sin interpretar (cuyo significado acuerden los procesos que se comunican).



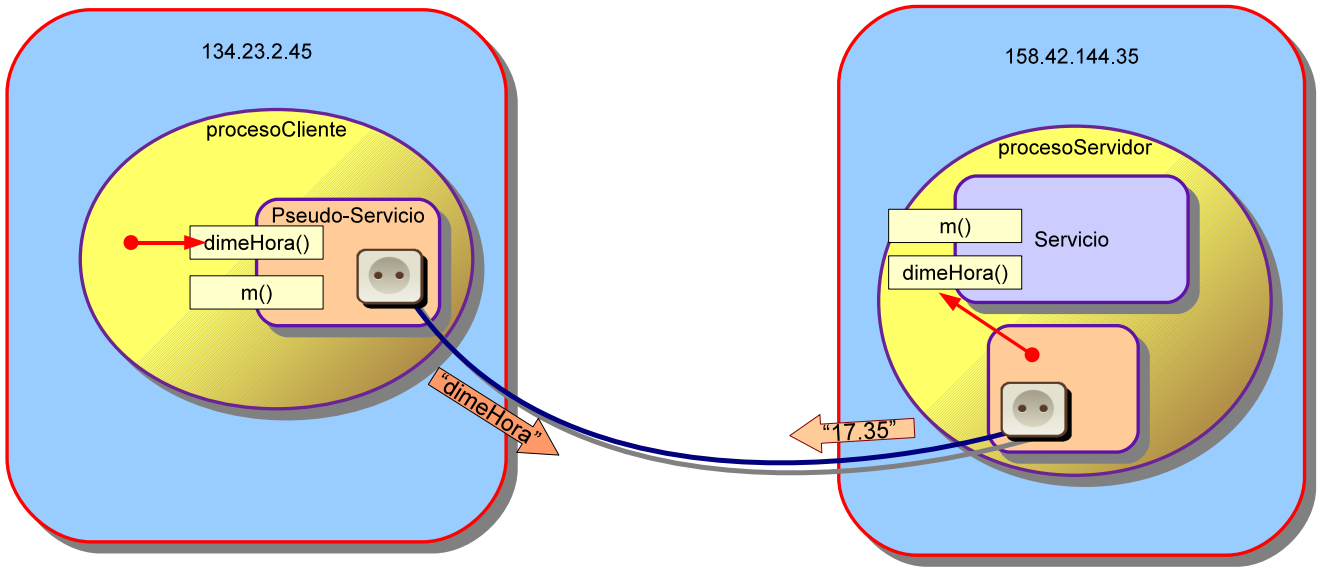
1.2.3 Llamada a procedimiento remoto

Serialización

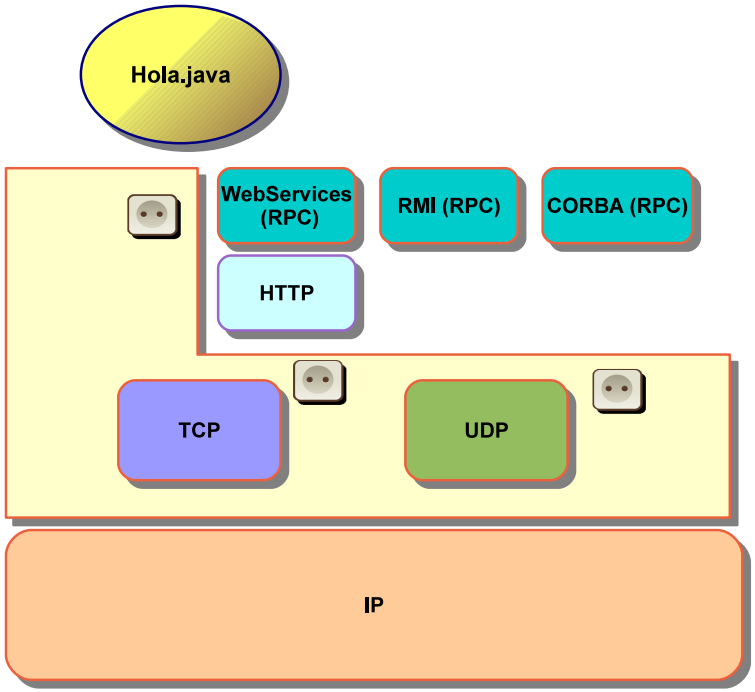
Hemos dicho que la comunicación utilizado flujos vinculados a sockets es muy conveniente, pero aún tiene una traba que la hace tediosa: la serialización y deserialización (marshalling, unmarshaling) de la información a transmitir. A través del flujo vinculado a un socket se puede enviar tanto texto como bytes sin interpretar. El problema viene cuando queremos enviar un valor de nuestro programa que no es de tipo texto. En principio se podría enviar como bytes, pero si el programa receptor no está escrito en nuestro lenguaje de programación, cosa que puede ocurrir perfectamente, no podrá recuperar el valor original a partir de los bytes que reciba. Por tanto, hay que convertir (*serializar*) el valor que se quiere enviar a un formato normalmente basado en texto, de forma que a su llegada, el receptor pueda reconstruir (*deserializar*) el valor original en su lenguaje de programación. Otro asunto relacionado (y aburrido) es la forma de invocación de las distintas funciones que un servidor tiene. Efectivamente, es normal que un servidor no realice una sola tarea. Cuando esto ocurre, es el protocolo entre el cliente y el servidor el que define qué debe enviar el cliente para solicitar un determinado servicio, qué parámetros debe enviar y qué en formato han de estar todos esos datos. Por tanto, en el cliente deberemos escribir código que construya la solicitud que enviaremos a través de socket. A su recepción, el servidor descompondrá la solicitud y acabará llamando a la función concreta para dicha solicitud.

RPC

Todo este proceso (construcción en el cliente y análisis en el servidor de las solicitudes) puede simplificarse mediante un bloque de software que *simule* una llamada a procedimiento remoto (*Remote Procedure Call, RPC*).



De esta forma, la programación del cliente y del servidor se facilita enormemente. El cliente para realizar una solicitud al servidor, llama a un método local, pero que en realidad redirige la petición, a través del socket, al servidor. Igualmente, para escribir un servidor, no tendremos que preocuparnos de la transmisión a nivel de socket, sino sólo de escribir métodos como hacemos normalmente, con la ventaja que podrán ser llamados desde un proceso cliente.



Hay distintos sistemas de llamada a procedimiento remoto:

- ONC RPC (*Open Network Computing Remote Procedure Cal*) utilizado originalmente en estaciones de trabajo UNIX.
- RMI (*Remote Method Invocation*) sistema propio de Java, para conectar procesos escritos sólo en Java.
- CORBA (*Common Object Request Broquer Architecture*) estándar pensado para intercomunicar procesos sin importar el lenguaje de programación de cada cual.
- Web Services. Estándar para conectar procesos remotos, basado normalmente sobre el protocolo HTTP[11]. Este RPC es actualmente uno de los más utilizados ya que se aprovecha de la popularidad e implantación ge-

[11] Aunque se podría teóricamente basar en otro como SMTP. En cualquier caso, no se basa directamente en sockets.

neralizada de HTTP, y gracias a ello permite interconectar clientes y servidores escritos en diferentes lenguajes de programación.
Originalmente, Web Services utiliza el estándar SOAP[12](*Simple Object Access Protocol*) que define el formato en que deben describirse los métodos remotos que pretendan invocarse, y cómo deben ser los mensajes que, internamente, se transmiten para soportar las llamadas remotas. Todas estas definiciones se basan en XML (*Extensible Markup Language*, una generalización de HTML).
Sin embargo, hay una tendencia a dejar de utilizar RPC Web Services basados en SOAP en favor de REST (*Representational State Transfer*). El estandar REST dicta, por ejemplo, que los tipos de llamadas que un cliente puede efectuar sobre un servidor sean *crear, leer, actualizar o borrar* (*create, read, update, delete* CRUD). La información transferida como argumento para la llamada debe estar codificada en el propio método de llamada (p. ej. GET hazCalculo?param1=5&&param2=8). La respuesta se devolverá en XML, pero a diferencia de SOAP, en un formato acordado por cliente y servidor.
Hay que hacer notar que los Web Services renuncian, para simplificar, a algunos objetivos que antes hemos marcado como deseables.

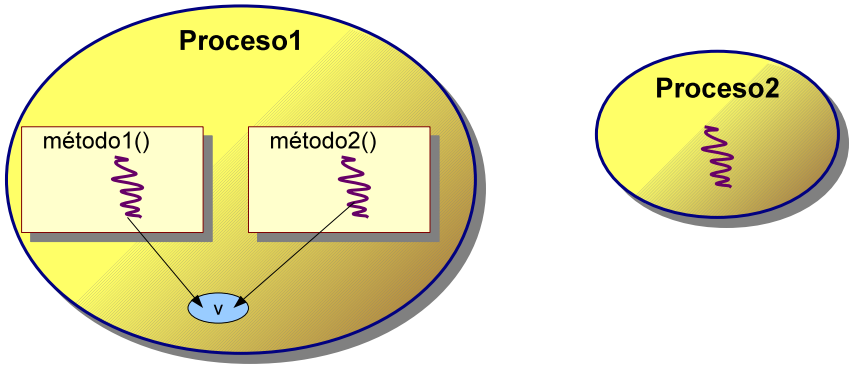
- La conexión permanente entre cliente y servidor. Los Web Services se basan en el protocolo HTTP, y aunque éste utiliza sockets TPC con conexión, se cierra dicha conexión al final de cada petición. Esto hace que en principio el servidor sea *sin estado* (*stateless*) y por defecto no tenga memoria de conexiones anteriores del mismo cliente.
- En el caso concreto de REST Web Services (no en SOAP Web Services), hay que acomodar en una operación CRUD todas funciones del servidor aunque alguna de ellas no se corresponda directamente con una operación CRUD. En esta situación perdemos la mecanización de una llamada remota para tener que realizar nosotros el formateo de los datos enviados y recibidos.

Todo este conjunto de software desarrollado con el propósito de interconectar procesos se conoce con el nombre de middleware ("software entre medio").

[12] Y aún antes, XML-RPC.

1.2.4 Hilos

Para terminar esta sección hemos de hablar de una última abstracción. Los hilos de ejecución (threads).



Por defecto, un proceso[13] ejecuta las instrucciones una tras otra, de forma que en un instante de tiempo sólo hay una instrucción ejecutándose. A bajo nivel, recordemos, existe un *contexto de ejecución* donde se guarda, especialmente, el *contador de programa* (registro que apunta a la posición de memoria donde está la instrucción que ha de ejecutarse). En el contexto de ejecución también se guardan los demás registros del procesador (puntero de pila, punteros a las zonas de datos del programa, etc.)

La cuestión es que, como sabemos, necesitamos que un ordenador haga varias tareas al mismo tiempo. Para ello hay dos opciones (que no son excluyentes entre sí):

- Tener varios procesos en ejecución.
- Ejecutar dentro de un proceso, al mismo tiempo, varias tareas. Por ejemplo ejecutar dos métodos al mismo tiempo.

[13] Proceso = programa en ejecución.

Las dos opciones tienen ventajas e inconvenientes contrapuestos. Si tenemos varios procesos en ejecución, como cada proceso es independiente, no comparten memoria. Esto significa que no habrá problemas con el acceso a variables compartidas, pero la comunicación entre ellos será más complicada.

Por otro lado, si en un mismo proceso se ejecutan dos tareas en paralelo, la comunicación entre ellas se hace fácilmente a través de variables comunes, pero hay que vigilar que esas variables no acaben en un valor inconsistente.

La opción que vamos a utilizar es la segunda: varias tareas ejecutándose en paralelo dentro de un mismo proceso. Cada tarea será realizada por un hilo de ejecución (*thread*). Un hilo de ejecución tiene en su contexto de ejecución un contador de programa diferente del de otro hilo, pero ambos compartirán los punteros a las zonas de datos, puesto que se encuentran en el mismo proceso.

Mediante el uso de hilos podremos escribir servidores que atiendan a varios clientes en paralelo[14].

Condiciones de carrera

El principal problema de programación con los hilos de ejecución son las condiciones de carrera (*race conditions*[15]). El ejemplo típico de condición de carrera se da cuando dos hilos leen el valor de una variable común y, un poco después, le suman 1. Como se ejecutan en paralelo, ambos hilos pueden haber leído el mismo valor. Si esto ocurre, también escribirán el mismo valor (el leído más 1), pero esto es incorrecto porque si no hubiera paralelismo la variable se habría incrementado en 2. Aquel trozo de código en donde pueden ocurrir condiciones de carrera se conoce con el nombre de sección crítica. Una sección crítica debe estar protegida de algún modo para evitar que dos threads entren al mismo tiempo en ella.

Sec. crítica

[14] Realmente, serán atendidos en paralelo sólo si el ordenador tiene un procesador con varios núcleos. Si no, el sistema operativo simulará la ejecución paralela cambiando de contexto de ejecución (y por tanto de proceso o de hilo en ejecución) cada cierto tiempo. Aún en este caso, es ventajoso utilizar hilos porque se puede amortizar el tiempo que un hilo pasa esperando que lleguen datos por algún flujo (teclado, disco, red).

[15] Condición de carrera es una mala traducción de race condition. Sería más acertado decir *problema de competición*.

2 Desarrollo de aplicaciones distribuidas en Java

2.1 Clases de biblioteca para ficheros

Recordemos que un fichero es una unidad de información almacenada en un disco u otro tipo de almacenamiento persistente[16]. Los ficheros se identifican con un nombre (texto). Los discos que contienen ficheros se organizan de forma jerárquica en carpetas (directorios).

En programación, se distingue entre:

- ficheros de texto. Aquellos cuyo contenido es interpretable según la tabla ASCII (son legibles por editores de texto). Se puede trabajar con ellos en base a líneas de texto y utilizando strings.
- ficheros binarios. Todos los demás: la interpretación del contenido[17] depende del programa que lo creó. Para trabajar con ellos utilizaremos un array de bytes.

Dento de un programa Java, un fichero se representa mediante un objeto de cierta clase. Las clases de biblioteca que suelen utilizarse al trabajar con ficheros en Java son las siguientes:

- `java.io.File`: representa un fichero en general
- `java.io.FileReader`: representa un fichero de texto del que leer.
- `java.io.FileWriter`: representa un fichero de texto en donde escribir.
- `java.io.FileInputStream`: representa un fichero del que leer.
- `java.io.FileOutputStream`: representa un fichero en donde escribir.

Las anteriores clases tienen métodos `read()` y `write()` para leer y escribir, caracteres (también bytes) de uno en uno, o agrupados en arrays (no como strings), lo cual hace pesado trabajar directamente con ellos.

[16] Unidad persistente: la que no pierde los datos al apagarla.
[17] jpg, gif, wav, mpg.

Afortunadamente hay otras clases que permiten:

- java.io.BufferedReader: leer strings con readLine()
- java.io.PrintWriter: escribir strings con println()
- java.io.DataInputStream: leer un fichero binario completo en una operación, leer valores de variables (int, double, boolean, etc.).
- java.io.DataOutputStream: escribir valores de variables (int, double, boolean, etc.).

La separación puede parerecer complicada e innecesaria pero no es así. El primer bloque de clases se refiere específicamente a ficheros, pero este segundo bloque no. Esto permite separar el origen o destino de los datos de las operaciones de conveniencia sobre los mismos, como leer o escribir strings. Es decir, podremos seguir utilizando las mismas clases y métodos para leer o escribir en algo que no sea un fichero (por ejemplo, un socket).

Al trabajar con ficheros, hay que tener en cuenta que puede haber problemas, no de programación, sino del entorno de nuestro programa (nombres de ficheros que no existen, discos protegidos contra escritura o ilegibles, etc.). Nuestro programa será avisado de esos problemas mediante las excepciones:

- java.io.FileNotFoundException
- java.io.IOException

Ejercicio

Lee el programa ficheros.UsarFicherosDeTexto.main(), ejecútalo y comprueba qué hace.

Aprende a ejecutar programas desde fuera de netbeans. Abre una consola y cambia al directorio Prog2Prac2/dist/ En ese directorio debe estar (después de haber compilado la práctica) el fichero Prog2Prac2.jar que contiene el programa entero. Ejecuta en la consola la orden[18]:

```
java -cp Prog2Prac2.jar ficheros.UsarFicherosDeTexto
```

Ejercicio

Lee el programa ficheros.UsarFicherosBinarios.main(), ejecútalo y comprueba qué hace. Utiliza un fichero que contenga una imagen.

Aprende a ejecutar programas desde fuera de netbeans. Abre una consola, cambia al directorio Prog2Prac2/dist/ y ejecuta en la consola la orden

```
java -cp Prog2Prac2.jar ficheros.UsarFicherosBinarios
```

Ejercicio

Escribe en ficheros.Main.main() un programilla que escriba tu nombre en un fichero de texto. Después, abre ese fichero con un editor para comprobar que lo has conseguido.

[18] Comprueba que el intérprete *java* está en el *PATH*, consulta con tu profesor.

2.2 Clases de biblioteca para hilos

Tener varios hilos de ejecución en Java es sencillísimo[19].

Primero necesitamos que una clase que implemente el interfaz Runnable, lo cual nos obligará a escribir un método run() en dicha clase.

```
public class Corredor implements Runnable {
  ...
  public void run () {
  }
}
```

Este método será el main() para los nuevos threads que creemos.

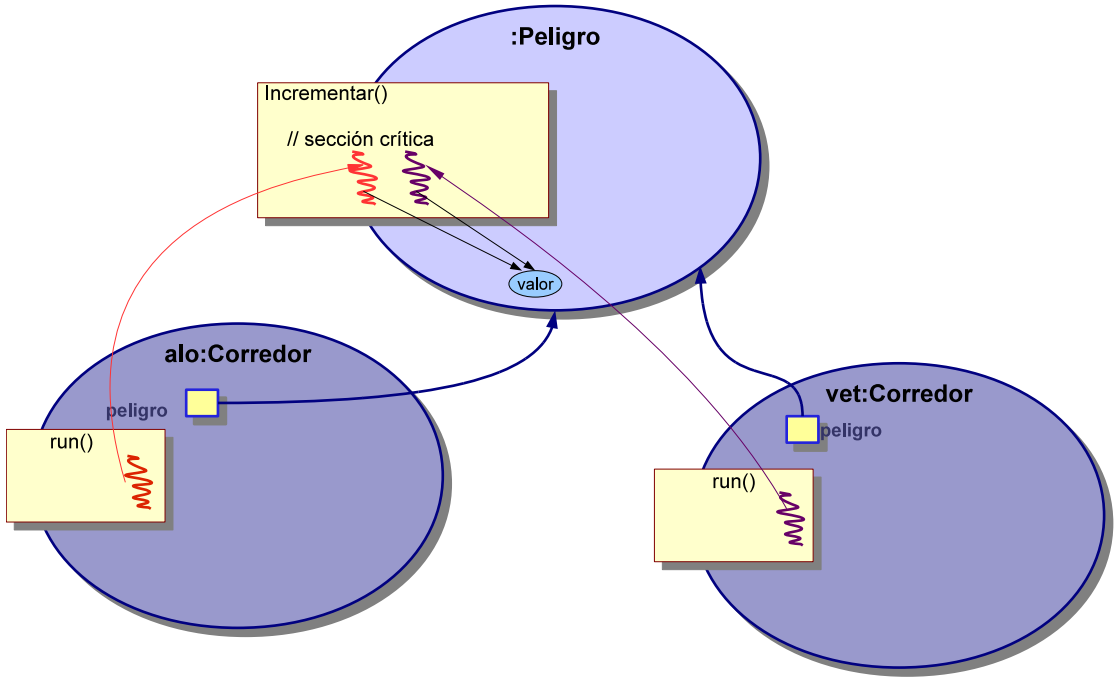
A continuación, en main() o donde nos convenga, crearemos objetos de tipo Corredor, y un objeto de tipo Thread para cada Corredor antes creado, y arrancaremos los threads con start(). Por ejemplo:

```
Corredor alo = new Corredor( ... );
Corredor vet = new Corredor( ... );

Thread th1 = new Thread(alo); // th1 ejecutará el run() de alo
Thread th2 = new Thread(vet); // th2 ejecutará el run() de vet

th1.start(); // arranca un thread en el run() alo
th2.start(); // arranca un thread en run() de vet
```

[19] El ejemplo sobre hilos se explicará también en clase de teoría.



Lo interesante del ejemplo que presentamos en el paquete hilos es comprender las condiciones de carrera que se podrían dar (si no se evita con synchronized) en la sección crítica que hay en el método Peligro.incrementar().

Ejercicio

Lee el código de hilos.Main, hilos.Corredor e hilos.Peligro.

Ejecuta hilos.Main.main() y comprueba qué ocurre cuando la sección crítica de hilos.Peligro.incrementar() esta protegida con synchronized(this) y cuando no.

Haz la ingeniería inversa de las anteriores clases.

2.3 Clases de biblioteca para sockets TCP

Recordemos que los sockets proporcionados por TCP son fiables y son orientados a la conexión. Por esto último, tienen asociados flujos de entrada y de salida, que permiten enviar o recibir información de forma similar a escribir en pantalla (println()) y a leer de teclado (readln()).

En concreto las clases relevantes son

- java.net.InetAddress: reprenta una dirección de internet
- java.net.ServerSocket: representa un socket de servidor
- java.net.Socket: representa un socket conectado

java.net.InetAddress permite, mediante métodos estáticos, y entre otras cosas

- InetAddress.getLocalHost().getHostAddress(): averiguar la dirección IP de nuestro ordenador
- InetAddress.getLocalHost().getHostName(): averiguar el nombre DNS[20] de nuestro ordenador.
- InetAddress.getByName("www.upv.es").getHostAddress(): averiguar la dirección IP del nombre DNS indicado
- InetAddress.getByName("158.42.4.23").getHostName(): averiguar el nombre DNS de la dirección IP indicada

[20] El protocolo *Domain Name System* permite convertir direcciones IP (158.42.4.23) en nombres de dominio más fáciles de recordar (www.upv.es) y viceversa.

2.3.1 Un servidor de internet

Un servidor de internet típico realiza el siguiente algoritmo:

```
ss ← nuevo socket vinculado a un numero de puerto conocido
repetir  por siempre
    Socket s ← conexión de un nuevo cliente aceptada en ss
    atender la petición (bien directamente o bien creando un nuevo hilo)
```

El anterior algoritmo, se corresponde en Java con el siguiente esqueleto:

```
// creo un socket de servidor
socketServidor = new ServerSocket(PUERTO);
// por siempre
while (true) {
  // 1. acepto conexiones (me paro si no hay)
  socketConectado = socketServidor.accept();
  // 2. atiendo la peticion:
  // 2.1 obtengo streams (flujos) para leer y escribir, asociados al socket conectado
  // (de esta forma puedo trabajar con lineas de texto completas)
          ...
  // 2.2 leo una linea de texto = RECIBIR
  linea = entrada.readLine();
  // 2.3 analizo linea para saber qué me piden
          ...
  // 2.4 contesto (ESCRIBIR la respuesta)
  salida.println( ... );
  salida.flush(); // que se envíe ya
```

```
      // 3. cierro los streams y el socket
               ...
      socketConectado.close();
    } // while true
```

Ejercicio

Lee el código del ejemplo echoTCP.Servidor.java

Ejecuta echoTCP.Servidor.main() desde una consola[21]:

```
java -cp Prog2Prac2.jar echoTCP.Servidor
```

Cuando arranca, el programa nos dice cuál es su dirección IP y en qué puerto está esperando.

Ejecuta[22] *en otra consola* (que puede estar en un ordenador distinto) la orden:

```
telnet 192.168.1.35 1234
```

Escribe algo y observa qué pasa en cada consola y relaciónalo con el programada del servidor: ¿Qué hace el servidor?

Prueba a escribir en un navegador cosas como http://192.168.1.35:1234 o http://192.168.1.35:1234/probando. ¿Qué ves? ¿Por qué?

[21] Cuando se ejecuta, el programa no termina nunca. En la consola utiliza CTRL-C para pararlo. En netbeans, detén el proceso en la barra inferior de la ventana.

[22] Cambia la dirección IP y el puerto por los datos que nos dice el servidor al arrancar.

2.3.2 Un servidor de internet paralelo

Cuando un servidor debe antender a una gran cantidad de clientes simúltáneamente, o el tiempo de atención de cada solicitud no es despreciable, debemos escribir un servidor paralelo: que pueda atender a varios clientes al mismo tiempo. Esto es sencillo utilizando threads en Java.

La técnica más elemental consiste en escribir una clase `public class HiloServidor implements Runnable` en cuyo método `run()` un nuevo thread atenderá a una petición concreta.

El constructor de esta clase recibe el socket conectado con el cliente `public HiloServidor (Socket con)` y se encarga de obtener los flujos que pueda necesitar para la comunicación y de arrancar el nuevo thread de `run()` haciendo

```
Thread th = new Thread (this);
th.start ();
```

Con esta nueva clase, el thread principal del servidor (el del `while(true)`) sólo tiene crear un nuevo objeto `HiloServidor` cada vez que reciba una petición.

Esta sencilla arquitectura es la que vamos a utilizar en la esta práctica. Pero, en un caso real, hay que completarla para amortizar el tiempo de creación de los threads y, sobre todo, para limitar la cantidad máxima de threads en ejecución, de forma que no se sobrecargue el servidor. Esto puede conseguirse con un *pool de threads*.

Ejercicio

Lee los ficheros `echoTCP.ServidorMultiHilo.java` y `echoTCP.HiloServidor.java`. Haz la ingeniería inversa de las anteriores clases.

Ejecuta en la consola:

```
java -cp Prog2Prac2.jar echoTCP.Servidor
```

Ejecuta[23] *en otra consola*:

```
telnet 192.168.1.35 1234
```

Escribe algo y observa qué pasa en cada consola y relaciónalo con el programada del servidor: ¿Qué hace el servidor?

Prueba a escribir en un navegador cosas como http://192.168.1.35:1234 o http://192.168.1.35:1234/probando. ¿Qué ves? ¿Por qué?

[23] Cambia la dirección IP y el puerto por los datos que nos dice el servidor al arrancar.

2.3.3 Un cliente de internet

El algoritmo de un cliente de internet es trivial:

```
s ← nuevo socket conectado a una dirección IP y puerto
enviar (escribir) la petición a través de s
recibir (leer) la respuesta de s
```

El anterior algoritmo, se corresponde en Java con el siguiente esqueleto:

```
// creo un socket conectándome a hostServidor PUERTO
Socket sock = new Socket(hostServidor, PUERTO);
// obtengo streams para leer y escribir a través del socket
...
// envio el mensaje (escribo)
salida.println(mensaje);
salida.flush();
// leo (recibo) la respuesta
String linea = entrada.readLine();
// cierro los streams y el socket
...
```

Ejercicio

Lee el código del ejemplo echoTCP.Cliente.java

Arranca alguno de los dos servidores que tenemos (Servidor o ServidorMultiHilo) como has hecho en los ejercicios anteriores. Ejecuta el cliente[24] en otra consola:

```
java -cp Prog2Prac2.jar echoTCP.Cliente 192.168.1.35 1234 saludos
```

Observa qué pasa en cada consola y relaciónalo con el programada del cliente y del servidor.

[24] Cambia la dirección IP y el puerto por los datos que nos dice el servidor al arrancar.

2.4 Desarrollo de un cliente y un servidor de ficheros

Vamos a escribir una sencilla aplicación distribuida que sirva para enviar ficheros de un ordenador (en el que ejecutaremos el servidor) a otro (el del cliente).

Para ello, primero debemos acordar un sencillo protocolo entre el cliente y el servidor. El protocolo que presentamos es una imitación minimalista del HTTP.

- El cliente (que siempre es el primero en hablar) enviará una solicitud en al servidor en una linea de texto con el siguiente formato:
  GET <fichero>
  por ejemplo:

  ```
  GET hola.txt
  ```

- El servidor responderá siempre con una línia de estado seguida de una línea en blanco. Si la solicitud es correcta, a continuación enviará los bytes del fichero pedido.
  Las lineas de estado, autoexplicativas, pueden ser las siguientes:
  - "HTTP/1.1 400 Bad Request : la solicitud no tiene 2 palabras"
  - "HTTP/1.1 400 Bad Request : la solicitud no es GET"
  - "HTTP/1.1 404 Not Found : no he econtrado el recurso"
  - "HTTP/1.1 200 OK"

2.4.1 El servidor

Escribiremos un servidor multi hilo aprovenchando un ejemplo anterior. Será muy sencillo porque el código para el servicio lo escribiremos en el método `servidorFicheros.HiloServidor.run()`. Será el siguiente:

1. Recibir la solicitud del cliente leyendo una linea de texto con
   `String linea = IO.leeLinea(this.entrada)`[25].
   Escribir la línea en pantalla como información.
2. Trocear (`String.split()`) la linea para sacar las palabras[26] que contiene:

   ```
   String[] trozos = linea.split("[ ]+");
   ```

3. Combrobar
   - que la línea de solicitud tiene al menos dos trozos. Si no, responder

     ```
     IO.escribeLinea("HTTP/1.1 400 Bad Request : la solicitud no tiene 2 palabras", this.salida);
     IO.escribeLinea("", this.salida); // después siempre se escribe una linea en
     blanco, tal y como hemos acordado en el protocolo.
     ```

     y terminar.
   - que la primera palabra de la solicitud es `GET`. Si no, responder `"HTTP/1.1 400 Bad Request : la solicitud no es GET"` y terminar.
4. Abrir el fichero pedido (segunda palabra de la solicitud)

   ```
   FileInputStream fich = new FileInputStream(trozos[1]);
   ```

   Al hacerlo, hay que capturar la excepción `FileNotFoundException` para, en caso que se dé, respoder `"HTTP/1.1 404 Not Found : no he econtrado el recurso"` y terminar.

[25] ¿Qué es `this.entrada` y `this.salida` si lo escribimos en `servidorFicheros.HiloServidor.run()`?
[26] Separadas por espacios en blanco.

5. Si hemos llegado hasta aquí ya podemos responder afirmativamente. Escribiremos "HTTP/1.1 200 OK", una línea en blanco y copiaremos los bytes del fichero del flujo de entrada fich al de salida (vinculado al socket):

```
int numBytes = IO.copia(fich, this.salida);
```

6. Ya está todo. Cerramos los flujos y el socket.

El método de utilidad

```
public static int copia (InputStream entrada, OutputStream salida) throws IOException
```

está incompleto. Su algoritmo es el siguiente:

```
int leido
int cont ← 0
leido ← byte sacado de entrada con .read()
repetir mientras leido ≠ -1
    escribir leido en salida con .write()
    cont ← cont + 1
    leido ← byte sacado de entrada con .read()
devolver cont
```

Ejercicio

Escribe el servidor según se ha descrito antes, así como el método IO.copia(fich, this.salida).

Ejecuta el servicio desde una consola

```
java -cp Prog2Prac2.jar servidorFicheros.ServidorMultiHilo
```

Aségurate que en el ordenador en que se ejecuta el servicio hay un directorio temporal llamado /tmp[27] con los ficheros contenidos en el adjunto html.zip

Realiza primero pruebas con telnet. En una consola escribe[28].

```
telnet localhost 1234

GET /tmp/hola.txt
```

Prueba tanto a pedir ficheros que existen en el directorio /tmp como a cometer errores (orden incorrecta, fichero que no existe) para comprobar las respuestas.

Realiza una pruebas desde un navegador, pidiendo ficheros que existan y ficheros que no. Por ejemplo:

http://192.168.1.35:1234/tmp/hola.txt

http://192.168.1.35:1234/tmp/hola.html

[27] c:\tmp en Windows si ejecutas el servidor en la unidad c:. Si no, cambia la letra de la unidad.

[28] Si servidor y cliente no están en el mismo ordenador, cambia localhost por la dirección IP del servidor. Ajusta también el número de puerto si no es 1234.

2.4.2 El cliente

Aunque como hemos visto, el servicio se puede utilizar con telnet o desde un navegador, vamos escribir el código de un cliente sencillo para aprender más.

Ejercicio

Hay que completar el código de servidorFicheros.Cliente.main() siguiendo las indicaciones que allí se dicen.

Para probarlo, en una consola prueba a escribir

```
java -cp Prog2Prac2.jar servidorFicheros.Cliente localhost 1234 /tmp/hola.txt
```

para que el servidor te envíe una copia del fichero /tmp/hola.txt

3 Entregas

3.1 Trabajo de lectura, pruebas, e implementación

- Antes de la primera sesión hay que
  - Leer la introducción (sección 1).
  - Hacer los ejercicios sobre ficheros de 2.1.
- Para la segunda sesión hay que
  - Tener ya preparado el ejercicio sobre hilos de 2.2
  - Empezar a hacer los ejercicios sobre sockets en 2.3
- Para la tercera sesión hay que preparar la implementación de la aplicación distribuida de copia de ficheros (sección 2.4) y implementarla durante esa sesión y la cuarta.

3.2 Código completo de la práctica

El fichero p2xxx.zip con el código completo se debe entregar electrónicamente en la propia tarea de poliformaT antes del 23 de abril de 2013

Jordi Bataller Mascarell 5 marzo 2013