Sistemas Distribuidos

Práctica 1: Sencillo servidor HTTP

Autor: Javier Celaya, modificada por Rafael Tolosana

Resumen

Esta práctica tiene como objetivo introduciros en la programación de aplicaciones cliente-servidor que gestionen múltiples clientes concurrentes, a través del conocido protocolo HTTP. Implementaréis un servidor web que ofrezca un pequeño subconjunto de las funcionalidades de HTTP en su versión 1.1, de manera que sea accesible desde un navegador web normal y corriente. Para atender peticiones de varios clientes de manera concurrente, existen principalmente dos maneras: utilizar un solo hilo que recibe información de todas las conexiones a la vez o un hilo por cada conexión.

Estas prácticas incluyen redactar una memoria y escribir código fuente. El texto de la memoria debe ser original, pero el código puede basarse en otro ya existente, siempre con el consentimiento del autor y citando la fuente. Por ejemplo, el código que encontréis en Internet puede ser utilizado siempre que citéis el nombre del autor y la URL de dónde lo habéis sacado. Sin embargo, las aportaciones personales siempre serán mejor valoradas. No citar la fuente se considera *plagio*, y plagiar supone un cero en la nota de prácticas.

Notas sobre esta práctica

- Recordad que al no tener privilegios, no podéis asignar un puerto menor de 1024 a vuestro servidor. Probablemente el cliente asumirá que el servidor está en el puerto 80, no olvidéis indicar el número correcto.
- No podéis usar las clases del paquete `com.sun.net.httpserver` para esta práctica.
- Vuestro servidor web devolverá el contenido de ficheros que se encuentren en el mismo directorio desde el que lo ejecutéis. No se os pide implementar ningún control de acceso, así que cuidado con la información que exponéis al exterior.

1. Gestión de múltiples conexiones en Java

En Java, la comunicación por red se hace a través de las clases en los paquetes `java.net` y `java.nio`. El primer paquete contiene las clases `Socket` y `ServerSocket` que os permitirán crear una comunicación cliente-servidor sencilla. La mayoría de ejemplos que encontraréis en Internet se basan tan solo en las clases de este paquete. Sin embargo, para hacer un servidor que maneje múltiples peticiones simultáneas deberéis usar las clases del paquete `java.nio`. Este paquete generaliza la comunicación por red y el acceso a ficheros a través del concepto de *canal*. Así, encontraréis los equivalentes `SocketChannel` y `ServerSocketChannel`, cuyo uso es muy similar, pero que permiten comunicación no bloqueante. La documentación oficial del paquete `java.nio`, junto con ejemplos, la encontraréis en [1]. Además, en Internet encontraréis multitud de tutoriales, como [2].

1.1. Comunicación bloqueante vs. no bloqueante

La forma más sencilla de gestionar una comunicación es con lecturas y escrituras bloqueantes. Una lectura de n bytes bloqueante significa "no quiero hacer nada más hasta que hayan llegado n bytes". Una escritura de n bytes bloqueante significa "no quiero hacer nada más hasta que se hayan enviado n bytes". La lógica del programa es más simple, pero ¿qué pasa si queremos seguir trabajando mientras se envían o se reciben datos? Para eso están las llamadas no bloqueantes. Una lectura de n bytes no bloqueante significa "dame todo lo que haya llegado, como mucho n bytes". Una escritura de n bytes no bloqueante significa "envía todo lo que puedas ahora, como mucho n bytes". Las operaciones no bloqueantes permiten seguir haciendo otras cosas mientras se llevan a cabo. A cambio, la lógica del programa es más compleja, porque su comportamiento es menos predecible. Debéis tener esto en cuenta cuando programéis el servidor web usando la llamada `select`.

1.2. La llamada `select` en Java

Anteriormente, habéis aprendido que el servidor sólo puede atender peticiones cuando está bloqueado en la llamada a `java.net.ServerSocket.accept()`. Una vez que ha aceptado una conexión entrante, no puede aceptar más mientras está ocupado gestionando esa petición. Para solucionar esto, existe la llamada `select` que habéis visto en clase. Su equivalente en Java es la clase `java.nio.channels.Selector`. En las dos referencias anteriores tenéis ejemplos de uso de esta clase.

A modo de recordatorio, la clase `Selector` se encarga de monitorizar el estado de una serie de canales, e informar de la ocurrencia de algún evento sobre ellos. Tras crear una instancia de la clase `Selector`, podéis registrar con ella cualquier número de instancias de `SocketChannel` y `ServerSocketChannel`. El objeto `Selector` os dirá cuándo un `ServerSocketChannel` está preparado para aceptar una nueva conexión entrante, y cuándo un `SocketChannel` está preparado para leer o escribir datos. El flujo de trabajo habitual sería el siguiente:

1. Registrar un objeto `ServerSocketChannel` con el `Selector` y esperar a que esté listo para aceptar una nueva conexión entrante. Esta nueva conexión genera una instancia de tipo `SocketChannel`, que se registra con el `Selector` a la espera de que existan datos para leer.

2. Cuando hay datos para leer en un `SocketChannel`, éstos se vuelcan en un buffer de tipo `java.nio.ByteBuffer`. Se procesan los datos y se calcula si se debe enviar una respuesta, *pero sólo se guarda*. Entonces se registra ese mismo `SocketChannel` a la espera de que se pueda escribir en él.

3. Cuando el `SocketChannel` está listo para escribir, se envía la respuesta.

Hay que tener en cuenta que un objeto `Selector` está pensado para trabajar con canales no bloqueantes. Esto tiene dos implicaciones:

- El `Selector` indica que hay datos para leer en cuanto recibe alguno. Puede que no hayan llegado aún todos los datos de la petición que envió el emisor. Debéis comprobarlo y, en ese caso, leer lo que haya disponible y esperar a que lleguen más datos.

- Cuando el `Selector` indica que se pueden escribir datos, puede que no os deje enviar toda una respuesta completa. De nuevo, debéis escribir todo lo posible y luego esperar a que el `Selector` indique que se pueden escribir datos de nuevo.

Esto permite intercalar al máximo las operaciones de entrada/salida con las de cálculo, de forma que se aprovecha mejor el tiempo.

¡Cuidado!. En lugar de leer todos los datos que llegan y escribir sólo cuando el objeto `Selector` dice que el canal está preparado, **es habitual caer en el siguiente diseño erróneo:** Al llegar los primeros datos a un canal, suponer que la petición ha llegado completa y procesarla, construir la respuesta y enviarla de vuelta como si se tratara de un canal bloqueante, todo de una sentada. Esta solución sólo os funcionará con peticiones y respuestas de unos pocos bytes. Además impide intercalar las operaciones, haciendo que vuestro servidor se comporte como si atendiera las peticiones de forma secuencial.

1.3. Hilos en Java

La alternativa a `java.nio.channels.Selector` para la gestión de múltiples peticiones concurrentes es lanzar un hilo nuevo para manejar cada nueva petición entrante. Normalmente crearéis un nuevo hilo con una clase que o bien implemente el interfaz `Runnable` o directamente se derive de la clase `Thread`. En ambos casos, debéis definir el método `void run()` con el código que queráis que se ejecute en un nuevo hilo. Aquí podéis utilizar llamadas bloqueantes, como siempre, ya que la máquina virtual de Java se encargará de intercalar unas con otras. La documentación oficial de Java tiene un buen tutorial sobre hilos y concurrencia en [3].

2. Objetivo de la práctica

Esta práctica tiene como objetivo que programéis un servidor web sencillo con dos implementaciones: una utilizando un sólo hilo y la clase `Selector`, y otra con un hilo independiente por cada petición. El servidor web debe cumplir los requisitos de la sección 3. Como resultado de la práctica entregaréis un fichero JAR que se llamará `SSDDp1.jar`. Tenéis más información sobre fichero JAR en [4].

El fichero JAR debe contener 2 cosas:

- una memoria, en la que resumiréis vuestro diseño del servidor, explicando cómo consigue gestionar múltiples conexiones simultáneas. La memoria tendrá una parte para cada una de las implementaciones del servidor. **Poned especial atención en explicar las funciones de comunicación, de manejo del objeto `Selector` y de manejo de hilos.** Ceñíos a lo que se os pide, sed claros y concisos. Además, debéis realizar y documentar una traza de pruebas para validar su correcto funcionamiento. Estas pruebas deben abarcar tanto los casos en los que se envía una petición correcta como incorrecta, mostrando las distintas respuestas del servidor. No os extendáis más de *3* páginas en total.
- debe contener todo el código fuente que hayáis desarrollado. Comentadlo conveniente, si fuera necesario. Poned atención a la organización del código, la modularidad, la legibilidad, la encapsulación, etc... Asegurándoos que cumple con las normas de estilo de Java.

Así, tiene que ser posible invocar el servidor web con la orden:

```
java -jar NIP_SSDDp1.jar -t|-s puerto
```

Se indicará el puerto en el que se quiere escuchar conexiones entrantes, y con la opción "-t" o "-s" se utilizará la implementación con hilos o con `Selector`, respectivamente.

3. Requisitos del servidor web

Vais a programar un servidor web que deberá implementar un subconjunto de características del protocolo HTTP v1.1 [5]. HTTP es un protocolo basado en peticiones y respuestas en texto plano, según el RFC 822 [6]. Las peticiones tienen la siguiente pinta:

```
metodo ruta HTTP/1.1
(Cabeceras...)
[Content-Length: longitud del cuerpo]
(Mas cabeceras...)

Cuerpo
```

La primera línea de la petición indica qué método se quiere realizar y sobre qué ruta. Las siguientes son cabeceras que dan información acerca de la operación. Una de ellas es `Content-Length`. Indica que la petición tiene un cuerpo y su longitud en bytes, incluyendo saltos de línea. Luego viene una línea en blanco que indica que comienza el cuerpo, y luego el cuerpo en sí, en caso de existir. Las respuestas son similares. La primera línea indica un código de estado, con información acerca de si la operación tuvo éxito o no, y por qué. Seguidamente vienen cabeceras que describen la respuesta y, opcionalmente, un cuerpo del mensaje con el resultado.

Concretamente, vuestro servidor **solo** debe aceptar los métodos GET y POST. El método GET devuelve el documento pedido y sus cabeceras, mientras que el POST permite enviar datos al servidor. Para cualquier otro método, se debe devolver un código 501 Not Implemented, y si la petición está mal formada se devolverá un código 400 Bad Request. Esto es suficiente para poder interactuar con el servidor desde un navegador web.

Estos métodos accederán a ficheros que se encuentren en el directorio actual. En caso de intentar acceder a un fichero del directorio actual que no exista, se devuelve un código 404 Not Found. En caso de intentar acceder a un fichero fuera del directorio actual, exista o no, se devuelve un código 403 Forbidden.

3.1. Tratamiento de peticiones

Completar

Para ayudaros a leer las peticiones de los clientes, se os proporcionan las clases HTTPParser y BlockingHTTPParser. Estas clases leen los datos que han llegado por un canal a un buffer de tipo `ByteBuffer`, en el primer caso, o por un `InputStream`, en el segundo caso, y los interpretan para extraer los datos relativos a una petición HTTP. El interfaz de estas clases consisten en los siguientes métodos:

```
class HTTPParser {
    public void parseRequest(ByteBuffer buffer);
    public boolean isComplete();
    public boolean failed();
    public String getMethod();
    public String getPath();
    public ByteBuffer getBody();
}
```

El método `parseRequest()` puede llamarse todas las veces que sean necesarias, pasando cada vez una parte adicional de los datos que componen la petición en un buffer de tipo `ByteBuffer`. Este método irá consumiendo los datos del buffer hasta que se detecte que la petición ha terminado. En ese momento, `isComplete()` devolverá `true` si la petición estaba bien formada, o `failed()` devolverá `true` en caso contrario. Si la petición estaba bien formada, `getMethod()` devuelve el nombre del método de la petición,

`getPath()` devuelve la ruta y `getBody()` devuelve el cuerpo si había, o `null` si no había cuerpo.

Es conveniente asociar una instancia de `HTTPParser` a cada conexión. Así, cada vez que un `SocketChannel` tiene datos para leer, esos datos se le pueden pasar al parser. Para asociar un objeto cualquiera a una conexión, la clase `SelectionKey` tiene los métodos `attach()` y `attachment()`.

3.2. Método GET

El método GET devuelve el contenido del fichero que se ha pasado como ruta. Las rutas son siempre relativas al directorio de trabajo del servidor web. Así, debéis comprobar que el fichero cumple los requisitos que se han mencionado antes.

Como ejemplo, suponed que ejecutáis el servidor en el directorio `/home/jcelaya`, donde hay un fichero llamado `foo.txt`. Abrís el navegador y en la barra de direcciones escribís algo como `http://vuestra.ip:puerto/foo.txt`. El navegador generará la siguiente petición:

```
GET /foo.txt HTTP/1.1
(cabeceras...)

```

Y vuestro servidor debería devolver un código 200 Ok, con el contenido del fichero en el cuerpo del mensaje:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 18

foo
bar
fooooooooo
```

La cabecera Content-Length es la única obligatoria en la respuesta, para que el cliente sepa cuándo termina el cuerpo del mensaje. Opcionalmente se pueden devolver otras cabeceras. En concreto, Content-Type es útil para indicar al navegador cómo debe tratar los datos que le enviáis: es una imagen, es un texto, es HTML...

La siguiente tabla os muestra ejemplos de peticiones erroneas, y sus correspondientes respuestas:

```
GET /bar.txt HTTP/1.1
```

(Suponiendo que el fichero `/home/jcelaya/bar.txt` no exista, claro...)

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: 90

<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
</body></html>
```

```
GET /media/picture.jpg HTTP/1.1
```

```
HTTP/1.1 403 Forbidden
Content-Type: text/html
Content-Length: 90

<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
</body></html>
```

```
GET /../../etc/passwd HTTP/1.1
```

```
HTTP/1.1 403 Forbidden
Content-Type: text/html
Content-Length: 90

<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
</body></html>
```

```
Me lo invento...
```

```
HTTP/1.1 400 Bad Request
Content-Type: text/html
Content-Length: 94

<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
</body></html>
```

```
HEAD /foo.txt HTTP/1.1

```

```
HTTP/1.1 501 Not Implemented
Content-Type: text/html
Content-Length: 102

<html><head>
<title>501 Not Implemented</title>
</head><body>
<h1>Not Implemented</h1>
</body></html>
```

3.3. Método POST

El método POST sirve para enviar datos al servidor web. Los datos van en el cuerpo del mensaje, y van codificados en el formato **application/x-www-form-urlencoded**. Este formato lo genera el navegador web a partir del contenido de un formulario HTML, y se decodifica fácilmente con la clase `java.net.URLDecoder`. Nosotros queremos usar el método POST para escribir datos en un fichero. Así, la ruta será siempre una barra (/), y el cuerpo del mensaje deberá ser así:

```
fname=nombre_fichero&content=contenido_fichero
```

Usad la clase `java.net.URLDecoder` para obtener el nombre del fichero en el que se debe escribir y el contenido. Respecto al nombre del fichero, debéis hacer las mismas comprobaciones que en el método GET y devolver el código de estado apropiado.

Para probar el método POST, guardad un fichero `post_form.html` en el directorio donde tengáis el servidor con el siguiente contenido:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head><title>Subir datos</title></head>
<body>
<h1>Subir datos</h1>
<form method=post action="http://vuestra.ip/">
Nombre del fichero: <input type="text" name="fname" /></br>
<textarea rows=5 cols=80 name=content></textarea></br>
<input type=submit value=Enviar /></form>
</body></html>
```

Al acceder a él con el método GET implementado anteriormente, el navegador os mostrará un formulario donde podéis introducir el nombre de fichero a guardar y el contenido. Por ejemplo, suponed que dais como nombre `foo.txt` y como contenido “Este es el contenido”. Al pulsar el botón “Enviar”, el navegador generará la siguiente petición:

```
POST / HTTP/1.1
(cabeceras...)
Content-Length: 42
(otras cabeceras...)

fname=foo.txt&content=Este es el contenido
```

Y vuestro servidor debería escribir ese contenido en el fichero `foo.txt` y devolver un código 200 Ok, con un pequeño mensaje de éxito:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 173

<html><head>
<title>¡Éxito!</title>
</head><body>
<h1>¡Éxito!</h1>
<p>Se ha escrito lo siguiente en el fichero foo.txt:</p>
<pre>
Este es el contenido
</pre>
</body></html>
```

4. Realización y Evaluación (Rúbrica)

La realización de las prácticas es por parejas, pero los dos componentes de la pareja deberán entregarla de forma individual. En general estos son los criterios de evaluación:

- Deben entregarse todos los programas, se valorará de forma negativa que falte algún programa / alguna funcionalidad.
- Todos los programas deben compilar correctamente, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas deben funcionar correctamente como se especifica en el problema.
- Todos los programas tienen que seguir el manual de estilo de Java propuesto por Oracle, disponible en moodle2. Además de lo especificado en el manual de estilo, cada fichero fuente deberá comenzar con la siguiente cabecera:

```
/*
 * AUTOR: nombre y apellidos
 * NIA: n'umero de identificaci'on del alumno
 * FICHERO: nombre del fichero
 * TIEMPO: tiempo en horas de codificaci'on
 * DESCRIPCI'ON: breve descripci'on del contenido del fichero
 */
```

4.1. Rúbrica

Con el objetivo de que, tanto los profesores como los estudiantes de esta asignatura por igual, puedan tener unos criterios de evaluación objetivos y justos, se propone la siguiente rúbrica en el Cuadro 2. Los valores de las celdas son los valores mínimos que hay que alcanzar para conseguir la calificación correspondiente y tienen el siguiente significado:

- A+ (excelente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea *correctamente* el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, *sin errores*. En el caso de la memoria, se valorará una estructura y una presentación adecuadas, la corrección del lenguaje así como el contenido explica de forma precisa los conceptos involucrados en la práctica. En el caso del código, este se ajusta exactamente a las guías de estilo propuestas.

- A (bueno). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea *correctamente* el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, *con ciertos errores* no graves. Por ejemplo, algunos pequeños casos (marginales) no se contemplan o no funionan correctamente. En el caso del código, este se ajusta *casi* exactamente a las guías de estilo propuestas.

- B (suficiente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No aplica el método de resolución adecuado y / o identifica la corrección de la solución, pero *con errores*. En el caso de la memoria, bien la estructura y / o la presentación son mejorables, el lenguaje presenta deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, este se ajusta a las guías de estilo propuestas, pero es mejorable.

- B- (suficiente, con deficiencias). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios

para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No se aplica el método de resolución adecuado y/o se identifica la corrección de la solución, pero *con errores* de cierta gravedad y/o sin proporcionar una solución completa. En el caso de la memoria, bien la estructura y / o la presentación son *manifiestamente* mejorables, el lenguaje presenta *serias* deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, hay que mejorarlo para que se ajuste a las guías de estilo propuestas.

- C (deficiente). El software no compila o presenta errores graves. La memoria no presenta una estructura coherente y/o el lenguaje utilizado es pobre y/o contiene errores gramaticales y/o ortográficos. En el caso del código, este no se ajusta exactamente a las guías de estilo propuestas.

Calificación	S.Threads	S.Select	Código	Memoria
10	A+	A+	A+	A+
9	A+	A+	A	A
8	A	A	A	A
7	A	A	B	B
6	B	B	B	B
5	B-	B-	B-	B-
suspenso	1 C			

Cuadro 2: Detalle de la rúbrica: los valores denotan valores mínimos que al menos se deben alcanzar para obtener la calificación correspondiente

5. Entrega y Defensa

Deberéis entregar un fichero zip que contenga: (i) los fuentes, (ii) el jar ejecutable y (iii) la memoria. La entrega se realizará a través de moodle2 en la actividad habilitada a tal efecto. La fecha de entrega será no más tarde del jueves anterior al comienzo de la siguiente práctica, esto es, el *15 de octubre de 2015* para los grupos A y el *22 de octubre de 2015* para los grupos B.

Los días 16 y 23 de octubre durante la sesión de prácticas se realizará una defensa "in situ" de la práctica.

Referencias

[1] *Documentación oficial del paquete* `java.nio`
`http://docs.oracle.com/javase/6/docs/technotes/guides/io/index.html`

[2] *Tutorial del paquete* `java.nio`
`http://tutorials.jenkov.com/java-nio/index.html`

[3] *Tutorial de concurrencia en Java*
`http://docs.oracle.com/javase/tutorial/essential/concurrency`

[4] *Empaquetar programas en archivos JAR*
`http://docs.oracle.com/javase/tutorial/deployment/jar/`

[5] *Especificación de la versión 1.1 del protocolo HTTP*
`http://www.w3.org/Protocols/rfc2616/rfc2616.html`

[6] *RFC 822: Standard for ARPA Internet Text Messages*
`http://tools.ietf.org/html/rfc822`