UNIVERSIDAD CARLOS III DE MADRID

Práctica 1: Comunicación de procesos usando sockets

Sistemas Distribuidos

100077162 Jorge Barata González

100072936 Cristian Galán Galiano

Índice

Diseño del programa

El programa se ha implementado usando el código base ofrecido para el desarrollo de la práctica, que sigue una arquitectura cliente-servidor. El cliente realizará diferentes peticiones al servidor, que le da respuesta.

De éste modo, la aplicación está dividida en dos componentes distintos: cliente y servidor. A continuación detallamos el diseño seguido para cada uno de ellos, así como los servicios implementados.

Cliente

El cliente inicia las peticiones, espera y recibe las respuestas del servidor a través de un socket TCP, usando la librería `socket`.

El programa recibe como parámetros la IP y puerto del servidor al que se desea conectar e intenta establecer conexión al principio de su ejecución.

Se ha implementado con una interfaz por consola, en la cual el usuario introduce los mandatos con sus parámetros. Como se puede ver en el diagrama de la Figura 1, una vez conectada la aplicación lee el mandato por la entrada estándar, lo interpreta y realiza la petición, procesando la respuesta y mostrando la información pertinente al usuario a través de la salida de error estándar. Esto se repite hasta que se ejecuta el mandato `quit`.

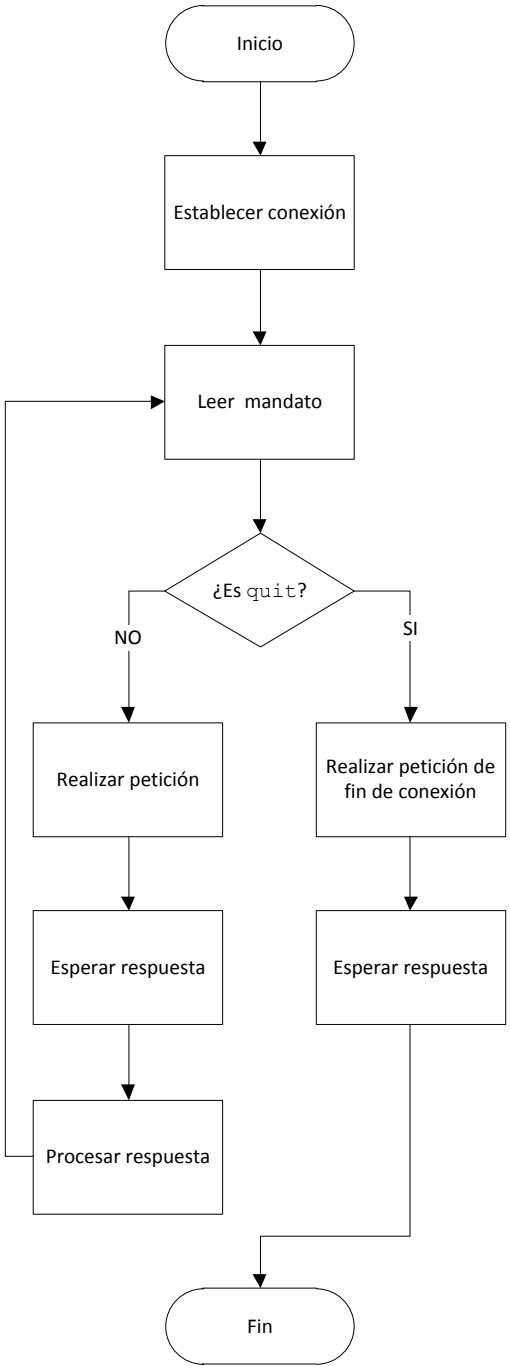

Figura 1: Diagrama de flujo del cliente

Servidor

El servidor es el receptor de las solicitudes del cliente. Su trabajo consiste en esperar y aceptar conexiones, esperar peticiones, procesarlas y devolver las respuestas. Al igual que el cliente usa sockets TCP.

El programa recibe un único parámetro: el puerto que se desea usar.

El servidor debe procurar sus servicios a un gran número de clientes simultáneamente, tal y como vemos en la Figura 2. Esto significa que tiene que ser capaz de mantener varias conexiones a la vez, así como de procesar las peticiones que lleguen a través de las mismas.

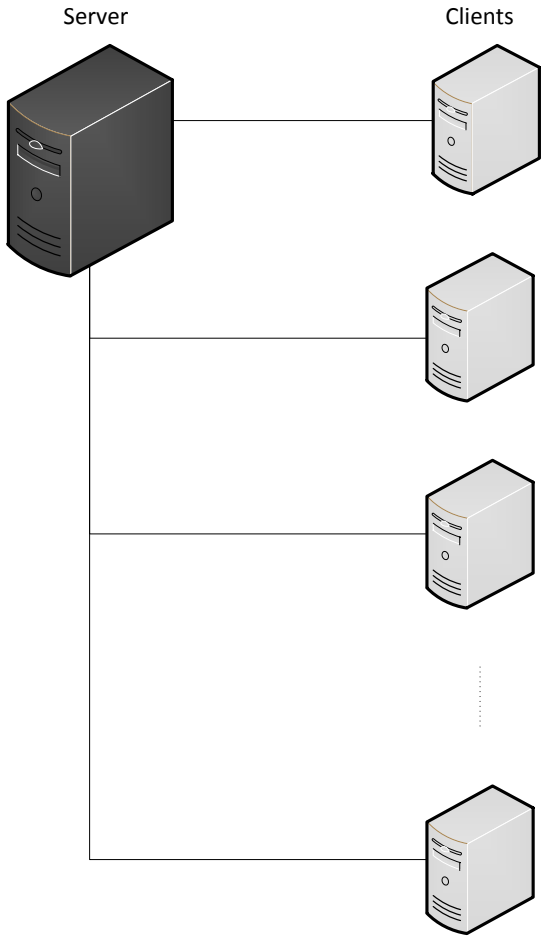

Figura 2: Arquitectura cliente-servidor

Para poder atender a múltiples clientes en el modelo de arquitectura cliente-servidor es común el uso de hilos de ejecución. En nuestro caso hemos usado procesos ligeros, por lo que además de la librería `socket` usada en el cliente, el servidor también usa la biblioteca estándar de hilos POSIX `pthread`.

Una vez corriendo, el servidor sólo usa la salida de error estándar para registrar las peticiones y no lee nada por la entrada estándar. Como vemos en la Figura 3, al iniciarse se queda escuchando en el puerto que se le haya indicado esperando peticiones. El servidor acepta las solicitudes de conexión y crea un hilo para cada una de ellas, de forma que cada cliente es escuchado y servido por hilos distintos. En la Figura 4 se muestra cómo son servidas las peticiones del cliente por los hilos.

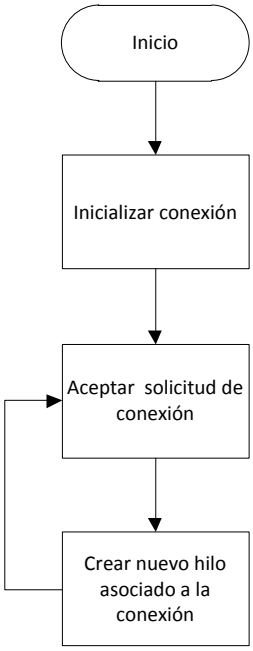

Figura 3: Diagrama de flujo del bucle principal del servidor

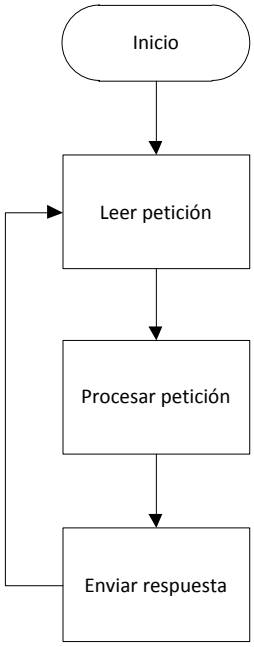

Figura 4: Diagrama de flujo del bucle de un hilo

El servidor tan sólo finaliza si sufre algún error, recibe una señal de interrupción o es matado. Entonces finaliza la conexión (Figura 5).

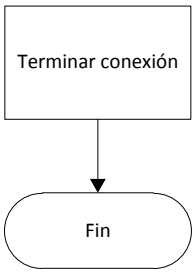

Figura 5: Fin de la ejecución del servidor

Servicios

Como en el enunciado ya se especifica la interfaz de los servicios ofrecidos tanto para el cliente como para el servidor, en éste apartado nos vamos a limitar a describir brevemente cómo se produce la comunicación desde que el usuario introduce un mandato hasta que recibe la respuesta. Asumimos que el cliente ya ha establecido conexión, por tanto el procesamiento de la petición es realizado por su hilo correspondiente en el servidor.

1. El cliente recibe el mandato
2. El cliente comprueba que el mandato y sus argumentos son correctos
3. El cliente realiza la petición al servidor
    a. El servidor informa al cliente de que ha recibido la petición. Esto se produce a nivel de TCP.
4. El servidor procesa la petición.
5. El servidor envía la respuesta al cliente.
    a. El cliente informa al servidor de que ha recibido la respuesta. De nuevo, hablamos a nivel TCP.
6. El cliente procesa la respuesta.

En la Figura 6 se puede ver el diagrama de secuencia:

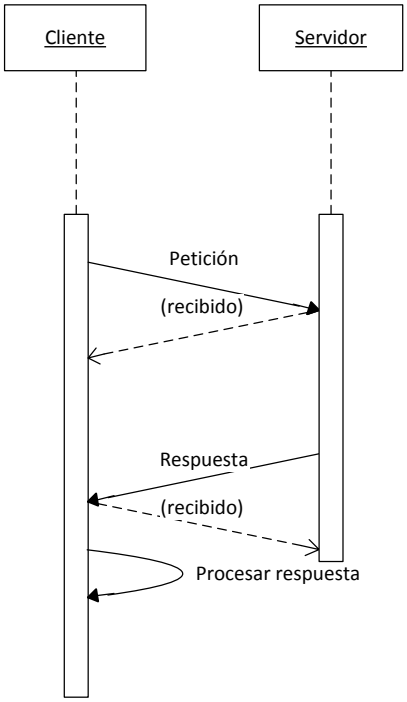

Figura 6: Diagrama de secuencia de la comunicación entre el cliente y el servidor

Esto se aplica para todos los mandatos, excepto para el mandato `quit`, el cual acaba en el punto 3 como podemos ver en la Figura 7:

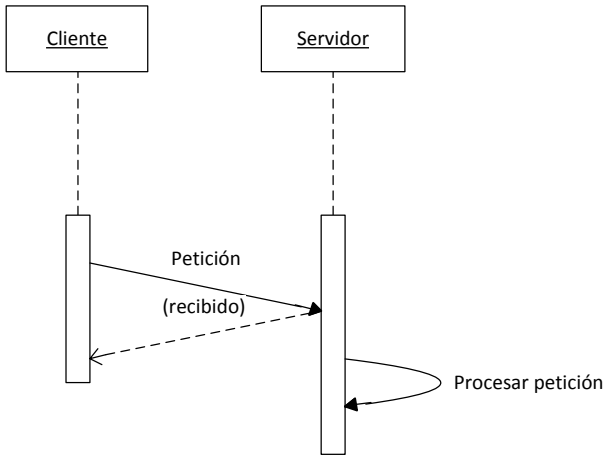

Figura 7: Diagrama de secuencia de la comunicación entre el cliente y el servidor para el mandato “quit”

Pruebas realizadas

Para la realización de las pruebas se ha usado el *script* de la Figura 8. Realiza lo siguiente:

1. Usa un artículo de la Wikipedia como fichero de pruebas.
2. Lanza el servidor en *background*, redirigiendo todas las salidas al fichero `server.out`.
3. Lanza cinco clientes en *background*, redirigiendo todas las salidas a ficheros diferentes para cada uno de ellos, llamados `client.out`. Para comunicarse con ellos posteriormente utiliza *named pipes*, comúnmente conocidas como *fifo*: son ficheros que mandan los datos recibidos de la entrada estándar directamente a la salida estándar. Imprimimos continuamente su contenido a la entrada estándar del cliente a través de *pipes*. Para enviar los mandatos basta con escribir en la *fifo* correspondiente al cliente con el que nos queramos comunicar.
4. Pedimos al primer cliente que calcule el *hash* del fichero usado para las pruebas para poder comprobar posteriormente que el *hash* es los servicios `hash` y `check` son correctos.
5. Ejecutamos una serie de mandatos para comprobar que todos funcionan correctamente. Lanzamos cada mandato como un proceso `echo` en *background*. De ésta forma provocamos una carrera entre los mandatos con el objetivo de simular un entorno real, con peticiones simultáneas en la medida de lo posible. En el *log* del servidor deberíamos apreciar como las peticiones han llegado en un orden diferente al que se lanzaron, así como peticiones que se han iniciado seguidamente antes de acabar las anteriores.

Hay que señalar que las solicitudes de establecimiento de conexión se efectúan secuencialmente. No las hemos incluido en la carrera de procesos del punto cinco dado que podrían mandarse mandatos antes de que los clientes se conectasen, llevando a error. Por la misma razón no hemos incluido el mandato `quit`, ya que los clientes podrían desconectarse antes de acabar todos los mandatos. Tuvimos que comprobar manualmente que los clientes podían conectarse y desconectarse en cualquier situación.

Al finalizar el script, tenemos los siguientes ficheros que nos sirven para comprobar que la aplicación funciona correctamente:

- Salidas de los clientes
  - `client.out.1`
  - `client.out.2`
  - `client.out.3`

    - client.out.4
    - client.out.5
- Fichero de prueba y fichero tras haber pasado por el servicio *swap*
    - file
    - file.swap
- Salida del servidor
    - server.out

```
#/bin/bash

SERVER=./server
CLIENT=./client
PORT=$(($RANDOM+1024))
WORKSPACE="../test"
CLIENT_FILE="$WORKSPACE/file"
CLIENT_FILE_SWAP="$WORKSPACE/file.swap"
CLIENT_FILE_URL="http://en.wikipedia.org/wiki/Distributed_computing"
CLIENT_INPUT="$WORKSPACE/client.in"
CLIENT_OUTPUT="$WORKSPACE/client.out"
SERVER_OUTPUT="$WORKSPACE/server.out"

#Preparamos el workspace
mkdir -p $WORKSPACE
rm $WORKSPACE/*

#Descargamos un fichero de ejemplo
curl $CLIENT_FILE_URL -o $CLIENT_FILE

#Iniciamos el servidor
$SERVER -p $PORT &> $SERVER_OUTPUT &

# Usamos una fifo para cada cliente
for n in {1..5}
do
      mkfifo $CLIENT_INPUT.$n
      tail -f $CLIENT_INPUT.$n | $CLIENT -s 0.0.0.0 -p $PORT  &>\
$CLIENT_OUTPUT.$n &
done

# Obtenemos el hash del documento con el primer cliente para\
comprobar las ejecuciones de check
```

```
echo "hash $CLIENT_FILE" > $CLIENT_INPUT.1 &
hash=`tail -n 1 $CLIENT_OUTPUT.1`

# Probamos los servicios.
commands=("ping" "swap $CLIENT_FILE $CLIENT_FILE_SWAP" "hash\
$CLIENT_FILE" "check $CLIENT_FILE 222" "check $CLIENT_FILE $hash"\
"stat")
for command in "${commands[@]}"
do
      for n in {1..5}
      do
            echo "$command" > $CLIENT_INPUT.$n &
      done
done

# Limpiamos las fifos
rm $CLIENT_INPUT*
exit
```

Figura 8: *Script* de pruebas

En la Figura 9 se muestra un fragmento de la salida del servidor en una de las ejecuciones con algunos comentarios. Recordaremos que el orden de secuencia de las peticiones de servicio es:

1. *hash* del primer cliente
2. *ping*
3. *swap*
4. *hash*
5. *check*
6. *stat*

```
s> init server 127.0.1.1:2246
s> waiting
s> accept 127.0.0.1:35607
s> waiting
s> accept 127.0.0.1:35608
s> waiting
s> accept 127.0.0.1:35609
s> waiting
s> accept 127.0.0.1:35610
```

Se inicia el servidor.

Acepta las solicitudes de conexión de los clientes.

```
s> waiting
s> accept 127.0.0.1:35611
s> waiting

s> 127.0.0.1:35607 init hash 19755
s> 127.0.0.1:35607 hash = 1682301

s> 127.0.0.1:35608 ping

s> 127.0.0.1:35611 ping

s> 127.0.0.1:35609 ping

s> 127.0.0.1:35610 ping

s> 127.0.0.1:35607 ping

s>127.0.0.1:35610 init swap 19755
s> 127.0.0.1:35610 swap = 13286

s>127.0.0.1:35611 init swap 19755
s> 127.0.0.1:35611 swap = 13286

s>127.0.0.1:35607 init swap 19755
s> 127.0.0.1:35607 swap = 13286

127.0.0.1> s:35609 init hash 19755
127.0.0.1> s:35608 init check 19755 222
s> 127.0.0.1:35609 hash = 1682301

s> 127.0.0.1:35608 check = FAIL

127.0.0.1> s:35610 init hash 19755
s> 127.0.0.1:35610 hash = 1682301

127.0.0.1> s:35611 init hash 19755
s> 127.0.0.1:35611 hash = 1682301

127.0.0.1> s:35610 init check 19755 222
s> 127.0.0.1:35610 check = FAIL

s>127.0.0.1:35609 init swap 19755
s> 127.0.0.1:35609 swap = 13286
```

Petición de *hash* del primer cliente para comprobación posterior.

Peticiones de *ping*.

Peticiones de *swap*.

Vemos cómo se inician una petición de *hash* antes y una petición de *check* simultáneamente: la segunda empieza antes de que acabe la primera. Además se han realizado antes de que acaben todos los *swap*. A partir de aquí se suceden las peticiones en un orden totalmente distinto al que se lanzaron.

```
127.0.0.1> s:35611 init check 19755 222
s> 127.0.0.1:35611 check = FAIL

s> 127.0.0.1:35611 init stat
s> 127.0.0.1:35611 stat = 1 1 1 1 0

127.0.0.1> s:35608 init hash 19755
s> 127.0.0.1:35608 hash = 1682301

127.0.0.1> s:35609 init check 19755 222
s> 127.0.0.1:35609 check = FAIL

s> 127.0.0.1:35609 init stat
s> 127.0.0.1:35609 stat = 1 1 1 1 0

127.0.0.1> s:35607 init check 19755 222
s> 127.0.0.1:35607 check = FAIL

s> 127.0.0.1:35607 init stat
s> 127.0.0.1:35607 stat = 1 1 1 1 0

s>127.0.0.1:35608 init swap 19755
```

Figura 9: Salida del servidor

En la Figura 1- vemos un ejemplo de la salida del cliente 1. Se puede apreciar que el orden de ejecución de los mandatos no es el mismo en el que se lanzaron. Esto significa que los mandatos se han mandado de forma distinta, pero de forma secuencial (no se debe confundir con el orden de los procesos concurrentes del servidor, que no se ejecutan secuencialmente).

```
1682301
0.000026 s
13286
FAIL
ping 1
swap 1
hash 1
check 1
stat 0
1682301
```

Petición de *hash* extraordinaria para comprobación posterior.
ping
swap
check
stat
hash

Figura 10: Salida del cliente

Conclusiones

La principal conclusión es que las aplicaciones distribuidas son posibles gracias a la programación multihilo, que permite procesar peticiones simultáneas, algo intrínseco en el cometido de un servidor en un modelo de arquitectura cliente-servidor.

Por otro lado, hemos visto que el control de errores es importantísimo en la depuración de una aplicación distribuida dada la complejidad del sistema, que depende de diversas capas de tecnología como red, sistema operativo y aplicación. No es nada trivial.

A pesar de todo, hemos apreciado la importancia del concepto de socket y su utilidad como herramienta en las comunicaciones TCP/IP entre programas.

Tareas realizadas

Tareas	Horas dedicadas
Documentación sobre *c*, *sockets*, hilos, etc.	7
Creación de conexión inicial	4
Implementar el servicio *ping*	2
Comunicar el cliente con el servidor usando el servicio *ping*	2
Implementar el servicio *intercambiar letras*	2
Comunicar el cliente con el servidor usando el servicio *intercambiar letras*	4
Implementar el servicio *calcular función resumen*	2
Comunicar el cliente con el servidor usando el servicio *calcular función resumen*	2
Implementar el servicio comprobar *función resumen*	1
Comunicar el cliente con el servidor usando el servicio *comprobar función resumen*	2
Implementar el servicio *obtener estadísticas*	1
Comunicar el cliente con el servidor usando el servicio *obtener estadísticas*	5
Envío de ficheros grandes	5
Pruebas	9
Memoria	8