Práctica 5: Programación en C++ de sistemas cliente–servidor basados en comunicación síncrona

Programación de Sistemas Concurrentes y Distribuidos

Dpto. de Informática e Ingeniería de Sistemas,
Grado de Ingeniería Informática
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

1. Objetivos

Esta práctica consiste en programar un sistema cliente–servidor mediante un modelo de comunicación síncrona, análogo al presentado en las clases de teoría. Los objetivos concretos de la práctica son:

- Programar procesos distribuidos que se comuniquen mediante *sockets* síncronos en C++.
- Aprender a implementar sistemas cliente–servidor en C++.
- Profundizar en el modelo de concurrencia de C++.

2. Trabajo previo a la sesión en el laboratorio

Antes de la correspondiente sesión en el laboratorio, cada pareja de estudiantes deberá leer el enunciado, analizar los problemas que en él se proponen y realizar un diseño previo de las soluciones sobre las que va a trabajar. *Los resultados de su trabajo de análisis y diseño los tendrá que expresar en un documento que entregará y presentará a los profesores antes del inicio de la sesión.* El documento debe contener como mínimo el nombre completo y el NIP de los dos estudiantes y, para cada ejercicio,

- la descripción de los procesos involucrados en el sistema y los protocolos de interacción que determinan la comunicación entre los procesos (contenido de los mensajes intercambiados, orden de los mensajes, gestión de errores en la interacción, etc.).
- un esbozo de alto nivel del código de los procesos cliente y servidor indicando las zonas que están afectadas por la comunicación y la sincronización.
- una descripción de los mecanismos utilizados en el lado del servidor para gestionar las restricciones de sincronización.

El documento deberá llamarse `informe_P5_NIP1_NIP2.pdf` (donde `NIP1` **es el NIP menor** y `NIP2` **es el NIP mayor** de la pareja), y deberá entregarse antes del comienzo de la sesión de prácticas utilizando el comando `someter` en la máquina `hendrix.cps.unizar.es`

```
someter progc_22 informe_P5_NIP1_NIP2.pdf
```

Además, una copia en papel de este documento deberá entregarse a los profesores al inicio de la sesión. Su entrega es un pre-requisito para la realización y evaluación de la práctica.

Por otro lado, el enunciado de la práctica también propone una serie de ejercicios previos con objeto de ayudar a los estudiantes a comprender cómo programar una aplicación cliente–servidor (sección 4). Se recomienda la realización de estos ejercicios antes de la sesión de laboratorio.

3. La abstracción de socket y el modelo cliente–servidor

Un modelo arquitectural fundamental para el desarrollo de aplicaciones distribuidas es el modelo *cliente–servidor*. Un *servidor* es un proceso que ofrece una o más operaciones que son accesibles a través de la red. Por otro lado, un *cliente* es un proceso que invoca (normalmente de forma remota) las operaciones ofrecidas por un servidor. El proceso servidor está a la espera de recibir invocaciones por parte de sus clientes. Una vez recibida una invocación, lleva a cabo el procesamiento requerido, usando los parámetros suministrados por el cliente, y le devuelve el correspondiente resultado.

La comunicación entre los procesos cliente y servidor requiere de una infraestructura de red. Una forma común de comunicación, presente en la mayoría de los entornos y lenguajes de desarrollo, es la basada en *sockets*. Los *sockets* representan los extremos de la conexión bidireccional que se establece para llevar a cabo esta comunicación. Cuando dos procesos requieren comunicarse solicitan al sistema operativo la creación de un *socket*. La respuesta a esta solicitud es un identificador que permite al proceso hacer referencia al nuevo *socket* creado.

Atendiendo a la pila de protocolos de Internet existen dos tipos de *sockets*:

- *Sockets* orientados a comunicación síncrona. El uso de este tipo de *sockets* proporciona una transmisión bidireccional, continua y fiable (los datos se comunican ordenados y sin duplicados) con conexión mediante el protocolo TCP (*Transport Control Protocol*).

- *Sockets* orientados a comunicación asíncrona. El uso de este tipo de *sockets* proporciona una transmisión bidireccional, no fiable, de longitud máxima prefijada y sin conexión mediante el protocolo UDP (*User Datagram Protocol*).

En esta práctica se proporciona una librería sencilla que abstrae los detalles de bajo nivel de comunicación vía sockets (la especificación de la librería se encuentra en los fuentes adjuntos en el web de la asignatura). La librería define una clase `Socket` con métodos públicos análogos a los utilizados en clase, enmascarando algunos detalles de manipulación de estructuras de bajo nivel, que estudiaréis en profundidad en otras asignaturas. Concretamente, en esta práctica nos centramos en comunicación síncrona entre procesos cliente y servidor.

Tanto para el proceso cliente como para el proceso servidor el primer paso será crear el objeto `Socket` mediante su constructor. Para su creación, el servidor tiene que suministrar el puerto en el que se va a publicar el servicio. Por su parte, el cliente tiene que suministrar tanto la IP donde se encuentra el servidor como el puerto del servicio.

Desde el punto de vista del **servidor** se requiere:

1. **Bind()**. Aviso al sistema operativo de que se quiere asociar el programa actual al socket abierto.
2. **Listen()**. Aviso al sistema operativo de que se procede a escuchar en la dirección establecida por el socket.
3. **Accept()** Aceptación de clientes. Si no hay ninguna petición de conexión, la función permanecerá bloqueada hasta que se produzca alguna. Cada vez que se ejecute, se acepta la conexión con un cliente que lo está solicitando.
4. **Receive()** Recepción de información a través del puerto. Análoga a la instrucción => utilizada en clase.
5. **Send()** Envío de información a través del puerto. Análoga a la instrucción <= utilizada en clase.
6. **Close()** Cierre de la comunicación y del socket.

En el **cliente**:

1. **Connect()**. Conexión con el servidor en la dirección y el puerto especificados.
2. **Send()** Envío de información a través del puerto.
3. **Receive()** Recepción de información a través del puerto.
4. **Close()** Cierre de la comunicación y del socket.

4. Ejercicios previos

El propósito de los dos siguientes ejercicios es introducir y formar al alumno en la programación de sistemas cliente–servidor. Se recomienda encarecidamente programar, ejecutar y comprender el funcionamiento de estos dos sencillos ejemplos como paso previo a la realización de la práctica.

4.1. Ejercicio 1

Los fuentes adjuntos en el web de la asignatura contienen un ejemplo de proceso servidor que escucha peticiones de un cliente en un puerto específico, `Servidor.cpp`), así como el código de un cliente que usa dicho servicio, `Cliente.cpp`. La instrucción

```
string SERVER_ADDRESS = "localhost";
```

obliga a que el cliente se ejecute en la misma máquina que el servidor.

El proceso servidor está asociado al número de puerto indicado en la variable `SERVER_PORT`. El proceso cliente solicita al proceso servidor que cuente el número de vocales de las frases que debe introducir el usuario por la entrada estándar. El proceso servidor atiende las peticiones del cliente y le comunica la respuesta, proceso que se repite hasta que recibe la cadena `END OF SERVICE`, momento en que el servidor finaliza su ejecución.

El cliente, en primer lugar, establece una conexión con el servidor para solicitarle sus servicios. A continuación, envía las diferentes peticiones de servicio, recibe las respuestas del servidor e informa al usuario (mediante la salida estándar) del número de vocales contabilizadas por el servidor para cada una de las frases introducidas. Cuando el usuario introduce la cadena `END OF SERVICE`, el cliente la remite al servidor y finaliza su ejecución.

Ejecutad el servidor en un terminal. A continuación, abrid un nuevo terminal y ejecutad el cliente. En este caso el servidor y el cliente se están ejecutando en la misma máquina; es decir, en modo local. Analizad el comportamiento de ambos procesos y las comunicaciones que se establecen entre ellos.

El ejercicio pide modificar ambos programas. En el caso del servidor, el puerto será un parámetro de invocación desde línea de comandos. En el caso del cliente, tomará dos parámetros: la IP donde se encuentra el servidor, así como el puerto en el que escucha. Para probar su ejecución, debéis averiguar la dirección IP del ordenador en el que vais a ejecutar el servidor (mediante el comando `host hendrix01.cps.unizar.es`, por ejemplo, o mediante `ifconfig` en linux, ...) y ponerlo en marcha. A continuación, ejecutad el cliente en un ordenador diferente al que ejecuta el servidor (por ejemplo, que el compañero que tengas al lado ejecute su cliente para que invoque el servicio que ofrece tu servidor). Analizad el comportamiento de ambos procesos y las comunicaciones que se establecen entre ellos.

4.2. Ejercicio 2

En la versión anterior el servidor queda asociado y conectado al primer cliente que se conecte. Lo habitual es que un servidor atienda simultáneamente a más de un cliente.

El fuente `ServidorMulticliente.cpp` desarrolla una variación del anterior para que el servidor atienda a múltiples clientes. Para ello, cuando recibe la petición de conexión por parte de un nuevo cliente (el servidor ejecuta un `Accept()`), crea un *thread*, adecuadamente parametrizado, para atender a ese cliente.

Ejecutad el servidor en un terminal. A continuación, ejecutad un cliente en local y otro en remoto y analizad el comportamiento de los procesos que se crean y las comunicaciones que se establecen entre ellos.

El ejercicio pide modificar ambos programas de la misma manera que en el caso anterior.

5. Trabajo a desarrollar

A continuación se describe el ejercicio a resolver durante esta práctica y las tareas que deben completar los alumnos.

5.1. Enunciado

Se trata de desarrollar una versión distribuida del sistema de las prácticas anteriores. Por un lado, tenemos que desarrollar un *profesor coordinador* y, por otro, un conjunto de 60 *estudiantes trabajadores*. El comportamiento, que debe seguir el esquema de protocolo mostrado en la figura 1, debe ser como sigue:

Estudiante

- Cada estudiante tiene una copia de la matriz
- Una vez se conecta al proceso coordinador, le envía el mensaje ‘‘`sentar,nip`’’, donde `nip` es su nip, único[1].
- Espera a recibir un mensaje con el contenido ‘‘`nipPareja,fila`’’, siendo `nipPareja` y `fila` el nip de su pareja y la fila con la que debe trabajar, respectivamente.
- Dependiendo de si su nip es el menor o no de la pareja, deberá calcular el máximo o la suma de la fila correspondiente.
- Deberá comunicar el resultado mediante el mensaje ‘‘`max,valor`’’ o ‘‘`suma,valor`’’, siendo `valor` el valor máximo de la línea o la suma de la fila, según corresponda al nip del estudiante.
- Ya puede terminar.

[1] El entrecomillado está puesto para delimitar el mensaje en este documento, pero **NO** forma parte de los mensajes.

Coordinador

- Lanzará un thread `Profesor` con el comportamiento del profesor, según la práctica anterior. Este thread terminará cuando todos los estudiantes hayan realizado sus tareas.
- Esperará a que los alumnos se vayan conectando, para poder encargarles una tarea
- Cada vez que un nuevo alumno se conecta lanza un thread representante para poder atender al alumno.

  Cada thread representante deberá entonces:

  - Esperar a recibir el mensaje ‘‘`sentar,nip`’’ de su estudiante representado.
  - Enviar el mensaje ‘‘`nipPareja,fila`’’ al representado
  - Esperar a que llegue la respuesta del representado que será o bien el máximo de la fila correspondiente o su suma, dependiendo de si el nip es mayor o no que el de su pareja
  - Si el nip asociado es el mayor de la pareja, deberá informar por la salida estándar de los valores obtenidos para la fila en cuestión, con el mismo formato que en la práctica anterior.
  - Terminará cuando haya realizado la tarea correspondiente

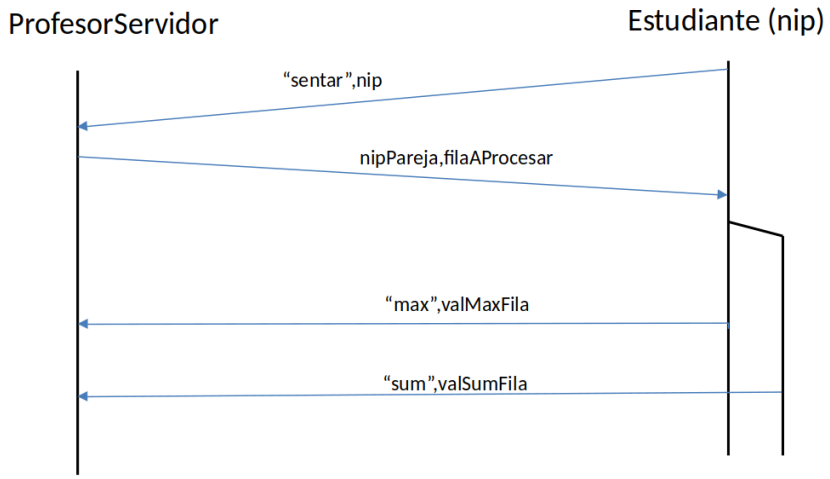

Figura 1: Esquema del protocolo a seguir.

5.2. Sobre la compilación en hendrix

Hendrix requiere librerías adicionales respecto a las requeridas en Linux, por ejemplo, en lo relativo a esta práctica. Estas son las que se nombran en la línea 28 del fichero `Makefile_p5`. Esa línea debe ser descomentada para compilar en hendrix.

6. Entrega de la práctica

Cuando la práctica se finalice, los dos componentes de la pareja deben entregar, cada uno desde su cuenta, el mismo fichero comprimido `practica_5_NIP1_NIP2.zip` (donde `NIP1` **es el NIP menor** y `NIP2` **es el NIP mayor** de la pareja) con el siguiente contenido:

1. Todos los ficheros fuente del coordinador programado.
2. Un fichero `Makefile_p5_COORDINADOR` que genera el ejecutable del coordinador, que se llamará `practica_5_COORDINADOR`.
3. El fichero `ejecuta_p5_COORDINADOR.bash` que:
   - compilará el código del coordinador usando `Makefile_p5_COORDINADOR`, generando el ejecutable del coordinador, que se pondrá en marcha mediante la ejecución siguiente, donde `<puerto>` es el puerto donde estará escuchando:

     ```
     ./Coordinador <puerto>
     ```

   - pondrá en marcha el coordinador escuchando en el puerto 3221
4. Todos los demás ficheros necesarios para que la ejecución de `Makefile_p5_COORDINADOR` genere el ejecutable requerido. Esto incluye una copia del directorio `Socket`.

Para la entrega del fichero `practica_5_NIP1_NIP2.zip` se utilizará el comando `someter` en la máquina `hendrix`. La fecha de entrega depende de la fecha en que se haya tenido la sesión de prácticas:

- Las sesiones del 30 de noviembre del 2022 deben entregar no más tarde del 7 de diciembre a las 08:00
- Las sesiones del 1 de diciembre del 2022 deben entregar no más tarde del 8 de diciembre, a las 08:00
- Las sesiones del 14 de diciembre del 2022 deben entregar no más tarde del 21 de diciembre, a las 08:00
- Las sesiones del 15 de diciembre del 2022 deben entregar no más tarde del 21 de diciembre, a las 08:00

Hay que asegurarse de que la práctica funciona correctamente en los ordenadores del laboratorio (vigilad aspectos como los permisos de ejecución, juego de caracteres utilizado en los ficheros, etc.). También es importante someter código limpio (donde se ha evitado introducir mensajes de depuración que no proporcionan información al usuario). El tratamiento de errores debe ser adecuado, de forma que si se producen debería informarse al usuario del tipo de error producido. Además se considerarán otros aspectos importantes como calidad del diseño del programa, adecuada documentación de los fuentes, correcto formateado de los fuentes, etc.

7. Evaluación de la práctica

La evaluación de esta práctica será presencial, en la fecha y hora establecidas con suficiente antelación por los profesores. Si un estudiante no se presenta a la evaluación, su práctica será calificada como "no presentada".

La pareja de estudiantes deberá ejecutar su servidor en una máquina cualquiera del laboratorio de prácticas en que se lleve a cabo la revisión, utilizando los fuentes y el script `bash` que sometió en las fechas previstas (se suministrarán en su momento las IPs de las máquinas a utilizar para poner los servicios). Durante la evaluación no se permitirá el uso de otros ficheros alternativos a los entregados, ni la modificación de estos. Por tanto, antes de someter la práctica os debéis asegurar de que el servidor programado es correcto y funciona y se ejecuta conforme a las pautas del enunciado.

Anexo

Ejemplo sencillo para separar componentes de una cadena según un carácter separador.

```
#include <sstream>
#include <vector>
...
//Pre:  <s> contiene un string con 1 o más campos, separados por el
//      carácter en <sep>
//Post: devuelve un vector cuyas componentes son los valores que
//      estaban separados por <sep> en <s>, por orden
//Com:  Mirar la documentación sobre la clase ''vector''

vector<string> split(const string& s, char sep) {
   vector<string> elems;
   string el;
   istringstream seqEls(s);
   while (getline(seqEls, el, sep)) {
      elems.push_back(el);
   }
   return elems;
}

//------------------------------------------------------
int main() {
    vector<string> trozos;
    string cad("RESERVAR 10,9,1,5");

    trozos = split(cad,',');
    for(int i=0; i<trozos.size(); i++) {
        cout << trozos[i] << endl;
    }
    return 0;
}
```