Programación Avanzada II

Práctica optativa:
LSTuit

Dani Álvarez Welters

La Salle – Ramon Llull
Grado en Informática
04/06/2015

Índice

Requerimientos

LSTuit es un programa con arquitectura cliente-servidor, en el que un mismo servidor permitirá la conexión de varios clientes simultáneos.

Para poder implementarlo necesitaremos dos ejecutables independientes, al tratarse de comunicación entre procesos en diferentes máquinas la única solución a fecha actual es usando sockets (comunicación a través de red).

Para garantizar la transmisión de datos y tal como se especifica en el enunciado (utilizando una comunicación orientada a la conexión), usaremos conexiones TCP, esto nos ayudará a que todo el contenido de los mensajes llegue íntegro y en orden temporal.
Usaremos entre otros sockets, hilos de ejecución distintos, lectura de ficheros, entrada/salida por pantalla (ver sección *Recursos utilizados*).

Diseño

La práctica se ha planificado de forma modular, de manera que se pueda reaprovechar diferentes partes.
El cliente y el servidor tienen dos programas principales independientes y referencian a unas librerías externas de tal manera que pueden aprovecharse ambos sin tener que repetir código.

Podemos encontrar:

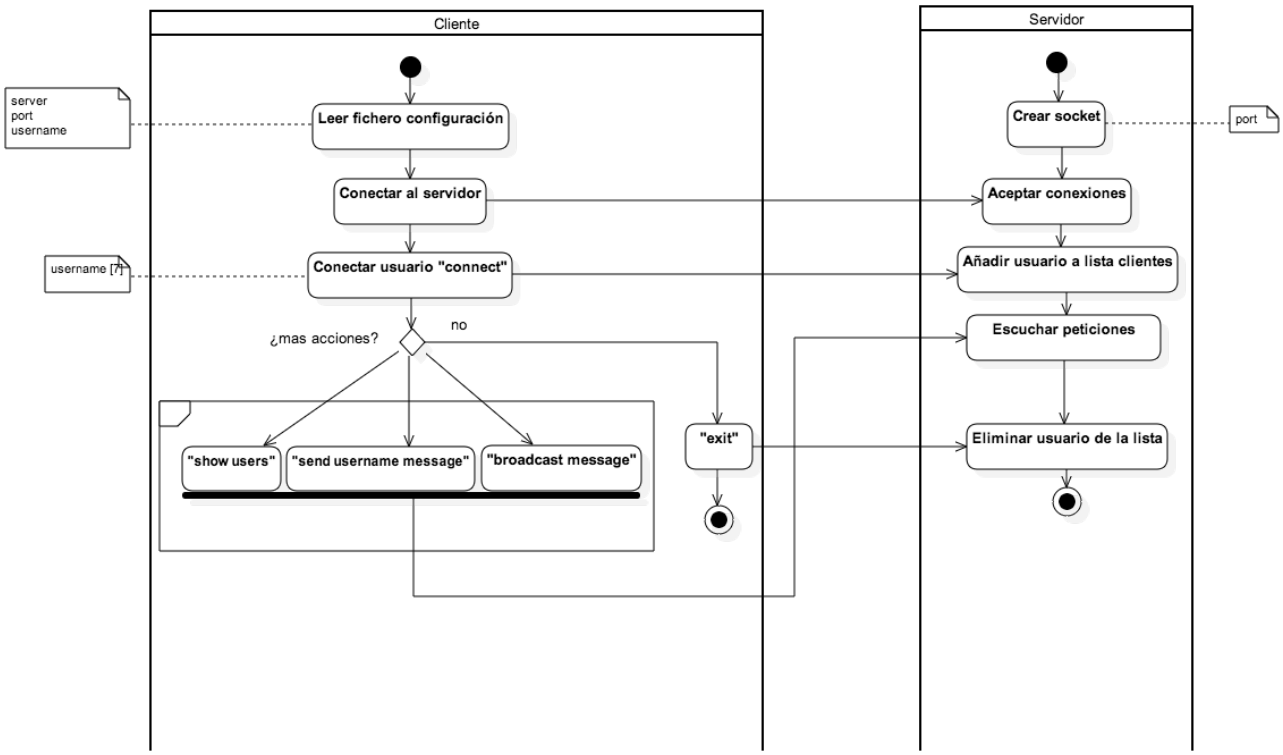
Servidor:
`server.c` Contiene el programa principal del servidor

Cliente:
`client.c` Contiene el programa principal del cliente

Librerías:
`user.h` Contiene la estructura de datos que guarda los usuarios conectados en el servidor y los métodos para añadir/quitar usuarios.
`data.h` Contiene la estructura de datos de las tramas a enviar
`error.h` Gestiona las excepciones y errores mostrados al usuario.
`logger.h` Librería de terceros para poder mostrar mensajes de log por pantalla o dirigirlos a un fichero de texto.

Diagramas

Para explicar el funcionamiento del programa he hecho un diagrama de actividad con el programa StarUML.



Como se han cumplido los requerimientos

Para cumplir los requisitos del enunciado se han creado diversas funciones.
A continuación detallo cada una y el uso aplicado, además de donde encontrarla dentro del código.

Cliente (`cliente.c`):

El programa principal del cliente lee el fichero de configuración que contiene la IP/nombre de host del servidor, el puerto al que debe conectarse y el usuario con el que se va a identificar.

Seguidamente se intenta abrir una conexión TCP al servidor mencionado anteriormente.

Una vez conectado, el programa principal levanta dos hilos de ejecución independientes, pasándoles como parámetro el file descriptor del socket a donde se ha conectado.

Cada hilo se dedicará paralelamente a recibir y enviar datos al servidor, el hilo de escribir envía datos al servidor siguiendo las especificaciones y el hilo de leer recibe la respuesta del servidor y muestra el resultado por pantalla.

Al escribir “exit” envía la señal al servidor para eliminar el usuario de la lista de clientes conectados y dejado de leer/escribir, liberando los recursos de memoria utilizados.

`int get_client_socket (const char *, const char *)`
Gestiona la conexión con el servidor. Usa la función getaddrinfo que combina internamente gethostbyname y getservbyname para obtener el primer destino correcto al que conectarse. Al encontrar un socket valido se conecta y retorna el file descriptor del socket al programa principal.

`void t_read (void *sock)`
Hilo dedicado a leer datos del servidor, dependiendo del tipo del mensaje recibido desde el servidor muestra los datos por pantalla.

`void t_write (void *sock)`
Hilo dedicado a leer de teclado las peticiones del usuario y “codificarlas” siguiendo las especificaciones y enviarlas al servidor.

Servidor (`server.c`):

Al ejecutar el servidor en primer lugar se comprueba que se haya pasado el argumento con el puerto donde se escucharán las peticiones.
Seguidamente *bindea* la conexión al puerto y abre la escucha.

Acepta todas las conexiones de clientes en cualquier interfaz de la máquina, para cada cliente levanta un hilo independiente pasándole una estructura que contiene el file descriptor donde se establecerá la comunicación con el cliente.

Dentro de cada hilo se llama a una función que se encarga de comunicarse con el cliente, leyendo del socket y dependiendo del tipo de mensaje recibido envía una información o otra según las especificaciones del enunciado.

Al recibir la instrucción de salida del cliente cierra el socket abierto.

```
int create_server_socket(unsigned short port)
```
Crea un socket TCP de escucha en todas las interfaces de la máquina en el puerto especificado por parámetros.

```
int accept_connection(int server_sock)
```
Al recibir el servidor una petición de conexión de un cliente acepta la conexión y retorna el file descriptor donde estará ese cliente. Después con el valor retornado se guarda en una estructura (thread_args).

```
void *thread_main(void *arg)
```
Cada vez que se recibe una conexión de un cliente y tenemos el file descriptor donde se encuentra, creamos un nuevo hilo pasándole dicho parámetro. En esta función obtenemos el identificador del hilo actual y lo pasamos a una función (handle_client) junto con el file descriptor del socket del cliente.

```
void handle_client(int client_socket, pthread_t thread_id)
```
Función que se dedica a escuchar todos los datos enviados por el cliente y devolver una respuesta con los datos necesarios.
Gestiona todo el envío de paquetes desde aquí, según el tipo de mensaje efectua una acción o otra.

```
int verifypackage(package *receive_msn)
```
Verifica el formato de todas las tramas recibidas por el cliente asegurando que ningún campo de la estructura llega vacío y el tipo de mensaje recibido existe.

Estructuras de datos usadas

Las estructuras de datos usadas son:

Package:
Usado para guardar las tramas de envío de mensaje, contiene el origen, el destino, el tipo de mensaje y los datos. El tamaño de cada campo viene definido por unas constantes siguiendo las especificaciones del enunciado.

```
char origin [NAME_SIZE]
char destiny [NAME_SIZE]
char type
char data [MESSAGE_SIZE]
```

User:
Usado para guardar la lista de usuarios conectados en el servidor, se trata de una lista simple enlazada. Contiene el nombre de usuario, el identificador del thread usado, el identificador del socket usado para comunicar y un puntero al siguiente usuario.

```
char name [7]
pthread_t thread
int sock
struct user * next
```

Thread_args:
Es un struct independiente para cada thread, va a almacenar su socket local, de forma que cada thread pueda manejar un socket independiente.

```
int client_sock
```

Recursos utilizados

Los recursos utilizados mas relevantes en este proyecto han sido:

- Sockets: Comunicar procesos en diferentes máquinas.
- Threads: Ejecutar procesos simultáneos.
- Mutex: Proteger recursos de memoria para asegurar la integridad de datos.
- Lista enlazada simple: Guardar la lista de usuarios conectados.
- Estructuras de datos: Almacenar datos de forma dinámica.

Otros recursos usados:

- Memcpy: Rellenar variables con un número exacto de datos.

- Makefile: Para hacer una compilación y ejecución más cómoda, el fichero makefile crea una carpeta bin con dos subcarpetas client i server, y genera un fichero de configuración config.dat para el cliente que por defecto se conecta a la dirección local en el puerto 1234 con usuario dani.
  También la opción *make clean* elimina las carpetas de ficheros ejecutables.

Pruebas realizadas

En primer lugar he probado a hacer las conexiones cliente-servidor en mi máquina local y la conexión se establece sin problemas.
Luego probando en el servidor Vela de la Salle, ambos programas ejecutarlos y también ha conectado satisfactoriamente, por último levantar el servidor en Matagalls y 3 clientes en Vela, también han conectado correctamente y se han podido comunicar.

Desde fuera de la red interna de la Salle no he podido comunicar cliente servidor, probablemente por restricciones de la universidad, para confirmar esto he usado un servidor externo y varios clientes en mi máquina local, también ha conectado.

Después he probado a quitar el fichero del configuración del cliente, iniciar servidor sin parámetro, nombre de usuario más largo de lo permitido, enviar mensaje más largo de lo permitido, enviar un mensaje a un usuario no existente.

Se ha permitido que conecten dos usuarios con mismo nombre, si se hace de esta manera si un emisor envía al usuario X abierto en dos clientes se entrega a ambos.

Relacionado con el envío de tramas, he abierto una sesión Telnet contra el servidor, intentando enviar datos corruptos para intentar que el servidor se caiga, no lo he conseguido.

Para comprobar la liberación de memoria al salir del programa he usado la herramienta *Valgrind*, que ya usé anteriormente en la otra práctica optativa, para confirmar que no queda memoria reservada.

Problemas encontrados

Inicialmente los problemas han sido conceptuales a la hora de diseñar el proyecto, probando con forks y memoria compartida para manejar los diferentes file descriptor de todos los sockets de clients se hacía complicado.
Así que finalmente lo cambié a threads, que no necesitaba de memoria compartida, y así asegurábamos que cada hilo independiente gestiona sus datos de forma independiente, de esta forma protegemos de alguna manera la integridad de datos.

La gestión de usuarios se ha hecho con una lista enlazada simple, al insertar dos usuarios a la vez no siempre funcionaba bien, por lo que se ha puesto un sistema de exclusión mutua.

Respecto al puerto abierto en la máquina a veces después de acabar el programa y haberlo cerrado, queda unos minutos inutilizables hasta que el sistema decide liberar el recurso y volverlo a poner disponible.

Finalmente, no como problema, pero si como tarea que se ha tenido en cuenta es liberar los recursos de memoria, en principio debería quedar todo liberado al finalizar el programa.

Conclusiones

Para concluir este proyecto, aunque se trate de algo compactado y con un enfoque muy académico (no usable en la vida real), me ha servido para repasar e implementar todos los conceptos trabajados a lo largo del curso, revisando la teoría proporcionada y el material adicional.

También he necesitado profundizar y buscar en Internet soluciones a las necesidades de implementación del programa.
El periodo para hacer el proyecto (1 mes aproximadamente) ha dado lugar a un resultado final muy aceptable donde comunicamos varios usuarios en diferentes puntos de internet a través de un servidor central.

Si hubiera tenido algo más de tiempo lo hubiera dedicado a efectuar un control de errores más exhaustivo y a implementar una interfaz mas amigable con *ncurses*.

Por encima de todo ha sido una experiencia positiva y entretenida.