Práctica Final

Sistemas Distribuidos

Alberto Penas Díaz:

NIA: 100471939

Correo: 100471939@alumnos.uc3m.es

Titulación: Ingeniería informática

Índice

Índice 1
Compilación y Ejecución 3
Descripción del Código. Decisiones de Diseño 4
Servidor 5
Concurrencia 5
Base de Datos 5
Cliente 6
Servicio RPC 7
Web Service 7
Batería de Pruebas 7
Evaluación de Rendimiento 9
Contenido Extra 10
Función Get Multifile 10
Recuperación de Errores 11
Analizador de Paquetes de Red 12
Conclusiones y Consideraciones Adicionales 12
Lagunas de Seguridad Importantes 12
Limitaciones en Red Local (UPnP) 12
Conclusiones Generales 13

Compilación y Ejecución

Para compilar y ejecutar los test del proyecto, se han elaborado scripts en bash que se encargan de elaborar todos los ejecutables de cada sección de la aplicación (la funcionalidad de estos ficheros así como las dependencias sobre otros programas de los mismos se han verificado en Guernika). Estos dos scripts son los siguientes:

- **app.sh**: este fichero se encarga de compilar y generar todos los ejecutables del proyecto. Para su mayor facilidad de uso, acepta los siguientes argumentos:
  - **-b (build)**: Construye todas las partes del proyecto que están escritas en C y por tanto, tienen que ser compiladas. Resumidamente y en orden, los pasos que sigue son los siguientes: 1. Genera los ficheros de código RPC con rpcgen 2. Compila el logger (servicio RPC) 3. Compila el servidor .

    Para la compilación de los ficheros en C, se ha utilizado la herramienta CMake, que facilita enormemente la vinculación de librerías y ficheros. Además, se han incluido todos los warnings[1] de compilación pertinentes para verificar la solidez del código y su correcta implementación. Estos warnings son los siguientes: -Wall -Wextra -Werror -pedantic -pedantic-errors -Wconversion -Wsign-conversion
  - **-c (clean)**: Limpia y elimina todos los ficheros objeto y rutas de compilación (cmake-build-release) que ha generado el build. También elimina los directorios *__pycache__* y los ficheros generados con el servicio rpc (stubs).
  - **-h (help)**: Muestra un mensaje de ayuda que explica el funcionamiento del script.
- **test.sh**: este script ejecuta todas las partes de la aplicación en procesos en segundo plano y procede a ejecutar todos los test (explicados en el apartado Batería de Pruebas). Este script necesita que el proyecto haya sido construido con antelación (con el comando ./app.sh -b) y no necesita ningún argumento.

Dados estos dos ficheros, compilar y ejecutar los test del proyecto sería tan sencillo como introducir los siguientes comandos en la terminal:

```
>> ./app.sh -b

>> ./test.sh
```

Para ejecutar cada componente por separado, primero se debe construir el proyecto con ./app.sh y después se debe ejecutar cada componente por separado, tal y como se muestra en el mensaje de finalización del fichero de compilación:

Componente	Comando para Ejecutar
Logger (Servicio RPC)	./logger/cmake-build-release/logger

[1] Los stubs generados por rpcgen no se han tenido en cuenta para la compilación con Warnings, ya que estos generan problemas ajenos dependientes de la herramienta que los genera (como variables que no se usan, etc). Es por ello que éstos ficheros se compilan SIN warnings.

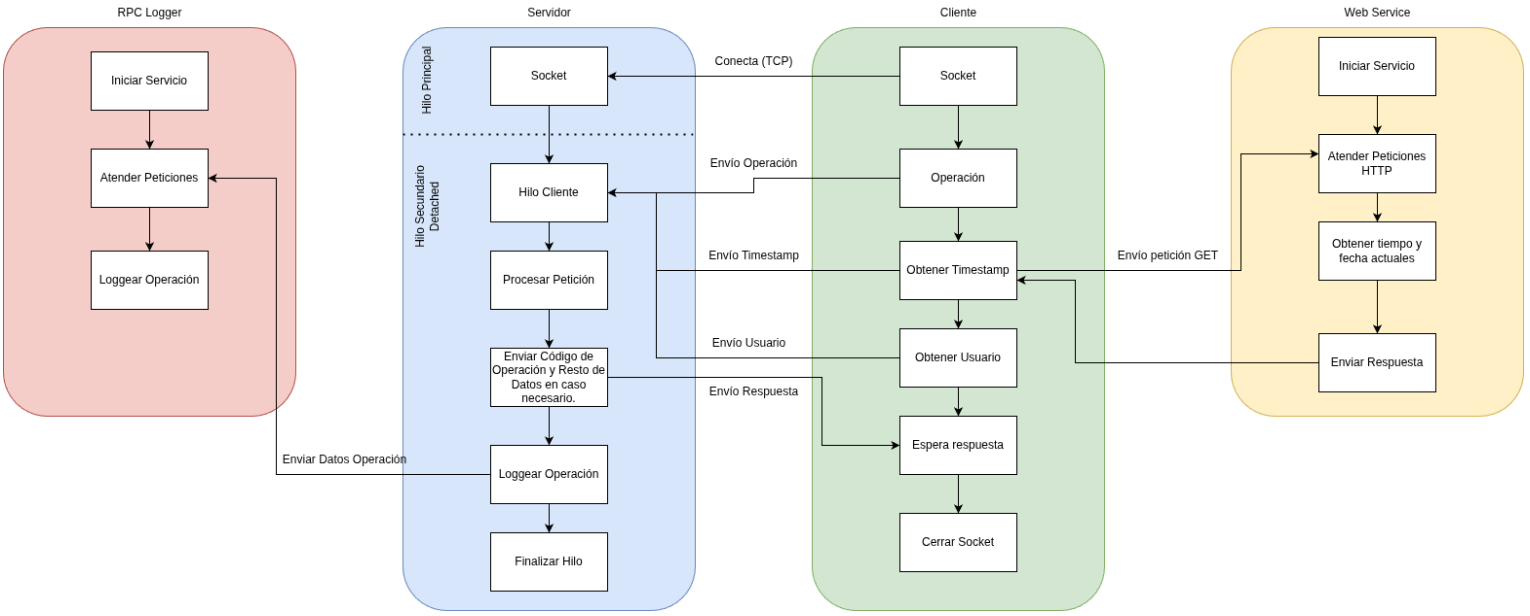
Servidor	env LOG_RPC_IP=<rpc_service_ip> ./server/cmake-build-release/server -p <port>
Web Service	python3 web_server/web_server.py
Cliente	python3 client/client.py -s <server_ip > -p <port>

Es importante aclarar que, tal y como se puede observar, todos los ficheros se ejecutan desde el directorio raíz, donde se encuentran los scripts de compilación y test. Esto es así porque se utilizan rutas relativas a la hora de importar librerías y paquetes. Es posible que haya problemas de ejecución si los componentes se compilan en una ruta diferente.

Cabe destacar también que cada componente "secuestra" la terminal en la que se ejecuta por lo que se recomienda ejecutar cada componente en segundo plano y redirigiendo la salida estándar a un fichero de logs.

Descripción del Código. Decisiones de Diseño

De forma esquemática, toda comunicación entre cliente y servidor se podría resumir en el siguiente diagrama:



Se ha omitido toda la información que el servidor puede llegar a responder dependiendo de la operación. Por ejemplo, en caso de que la operación sea REGISTER o UNREGISTER, el servidor solamente tendría que enviar de vuelta un código de error o éxito, mientras que si la operación en LIST_USERS o LIST_CONTENT, éste tendría que envíar mucha más información, tal y como se especifica en el enunciado de la asignatura. Como se ha seguido el protocolo del enunciado a rajatabla, se ha decidido omitir esta información por simplicidad y claridad en el diagrama.

Un aspecto importante a tener en cuenta es que el servidor se conecta con el servicio de logging RPC cuando ya ha finalizado todo el cómputo de la operación, esto es debido a que de esta manera, el cliente no ha de quedarse esperando a que la operación se loggee correctamente en un servicio que, a ojos del cliente, no le afecta en absoluto. De esta manera, el cliente puede seguir funcionando correctamente esté el cliente activo o no, pudiendo hacer más peticiones si así lo considera oportuno.

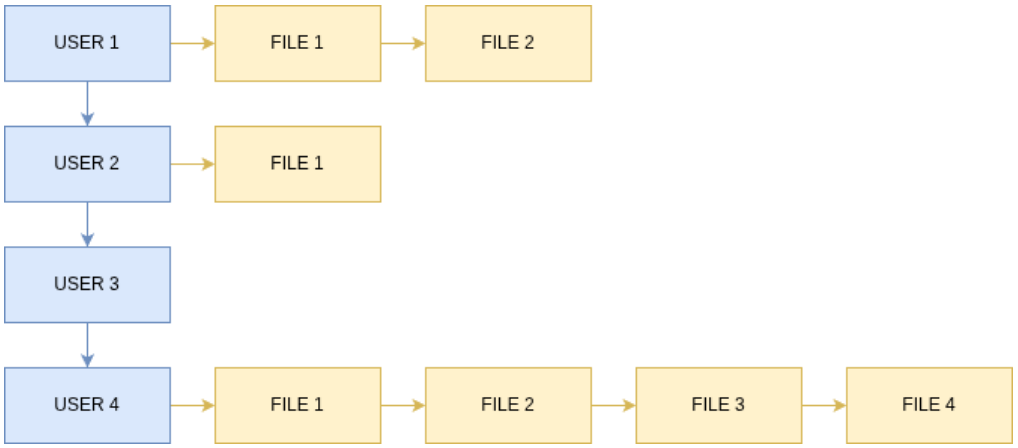
Servidor

Concurrencia

Para la parte del servidor, se ha seguido un modelo muy similar al de el segundo ejercicio de la asignatura. De forma resumida, el servidor escucha peticiones en el hilo principal. Por cada petición entrante en el socket de escucha principal, despliega un hilo detached a demanda. El hilo es detached por que no es necesario obtener ningún valor de retorno hacia el hilo principal desde el hilo secundario, ya que es éste último el que se comunica directamente con el cliente. Por cada petición nueva, se establece un mutex para poder evitar sobreescribir en la variable que se utiliza para determinar el descriptor del socket del cliente. Una vez que el hilo ha hecho una copia local de esta variable, la misma se libera para que pueda ser reutilizada por otros hilos.

Base de Datos

El fichero claves.c es el que se encarga de realizar todas las operaciones de guardado en memoria, búsqueda en el sistema, eliminación de ficheros y usuarios, etc. Básicamente, esta base de datos existe en RAM en una lista enlazada para los clientes, donde cada uno de estos tiene a su vez una lista enlazada para los ficheros. Ésta jerarquía se podría representar de la siguiente manera:



Cada usuario está compuesto por una estructura en C con los siguientes atributos:

- Nombre
- ip
- Puerto
- Conectado (variable booleana para saber si un usuario está conectado o no)
- lista de ficheros
- Puntero al siguiente usuario

De la misma manera, cada fichero está compuesto por una estructura en C con los siguientes atributos:

- Ruta
- Descripción
- Puntero al siguiente fichero

Por último, es importante mencionar que el fichero de claves.c, utiliza un mutex global para verificar que no se esté modificando una de estas estructuras al mismo tiempo. Este mutex bloquea a cualquier otro hilo que quiera acceder a cualquiera de las listas enlazadas si ya está siendo accedida por otro hilo. Sin embargo, para acelerar muchísimo la eficiencia de este sistema, se ha empleado un mutex especial: un **Read Write Mutex**, este tipo de mutex permite hacer distinción entre aquellas operaciones que van a ser solo de lectura y aquellas que van a ser de escritura (que son las que realmente generan condiciones de carrera). En este caso, si por alguna razón todos los usuarios conectado están haciendo operaciones de lectura, el sistema es inteligente y no bloquea de ninguna manera la lista enlazada, ya que no hay nadie que esté haciendo modificaciones a la misma. En cuanto haya un único hilo que quiera realizar una operación de escritura sobre la base de datos, automáticamente se bloquea el mutex tanto para la escritura como para la lectura. Este mutex se auto inicializa en su declaración.

Cliente

El cliente es completamente autónomo y puede funcionar sin necesidad de que el servicio web esté activo, lo que facilita la depuración y disminuye las dependencias externas, tal y como se ha comentado previamente.

El cliente admite tanto un modo interactivo como un modo no interactivo (desde fichero de entrada), lo que ha permitido su integración directa en la batería de pruebas automatizadas. Internamente, cada operación disponible (REGISTER, CONNECT, PUBLISH, etc.) se implementa como una función estática dentro de la clase client, manteniendo un diseño limpio y modular.

Uno de los aspectos más relevantes de la implementación es el tratamiento del estado del usuario. El cliente gestiona internamente el usuario actualmente conectado y lanza un hilo de escucha (ServerThread) cuando se realiza la conexión. Este hilo permite recibir peticiones entrantes de otros usuarios (por ejemplo, durante una descarga) y se cierra automáticamente al desconectarse. Con respecto a esto, es importante mencionar que **no se permite más de un usuario conectado al mismo tiempo** pero sí se permite que desde un mismo proceso, se registre a varios usuarios.

Cada mensaje enviado al servidor sigue el formato especificado en el enunciado, utilizando cadenas terminadas en \0 y gestionando las respuestas a través de códigos de estado enviados como un único byte (lo que permite despreocuparse por el "*endianess*"). Además, como pequeño punto extra, en la función PUBLISH, se ha considerado conveniente comprobar la existencia del fichero antes de poder publicarlo.Y también, por comodidad, es posible publicar un fichero especificando únicamente su path relativo, ya que el propio cliente se encarga de transformarla a path absoluto antes de publicarla en el servidor.

Por último, se han introducido validaciones robustas en todos los comandos para garantizar que los parámetros (longitud de nombres, rutas, existencia de ficheros, etc.) sean correctos antes de enviarlos

al servidor. También se ha automatizado la conversión a rutas absolutas y se han evitado rutas con espacios en blanco, mejorando así la fiabilidad del sistema.

Servicio RPC

La arquitectura del servicio de Logging con ONC-RPC está diseñada para que el servidor invoque este servicio de manera asíncrona: es decir, una vez que se ha completado el procesamiento de una petición del cliente, el servidor realiza una llamada RPC al logger para registrar la operación. De esta manera, se evita que el cliente tenga que esperar a que la operación sea registrada, ya que dicho logging no es relevante desde el punto de vista del cliente. Esta decisión de diseño mejora la experiencia del usuario y desacopla la lógica del servidor del sistema de registro.

El logger presenta una interfaz muy sencilla, recibiendo como único argumento una estructura con la información necesaria para realizar el string que se loguea por terminal en el servicio. En esta parte, el servidor de la aplicación actúa como cliente del servicio RPC y como la velocidad del servidor tiene que ser la máxima posible, solo se crea el cliente RPC si no existe previamente, reaprovechando al máximo las operaciones realizadas con anterioridad y evitando tener que estar creando el cliente RPC constantemente.

Web Service

El servicio web ha sido implementado en Python utilizando la librería *pysimplesoap* (Disponible en Guernika). A pesar de su simplicidad, el servicio cuenta con un diseño robusto: la función *get_datetime* se encuentra registrada en el dispatcher SOAP y expuesta mediante el estándar WSDL, permitiendo su invocación remota conforme a dicho protocolo. El cliente incluye una lógica de reconexión en caso de que el servicio no esté disponible inicialmente, permitiendo así que su funcionalidad no quede comprometida si este componente no está activo en el momento del arranque.

Batería de Pruebas

Para probar la funcionalidad del sistema, se ha realizado una extensa batería de pruebas para comprobar la funcionalidad de todas las posibles interacciones del usuario con la aplicación. Todas estas pruebas se ejecutan mediante el fichero test.sh y se puede observar por terminal si cada prueba ha sido exitosa o no.

Cada prueba tiene tres ficheros clave:

1. **Fichero de Entrada** (test_files/input/): Fichero en el que se especifican los comandos de entrada que simulan a un cliente. Un ejemplo de fichero de entrada puede ser el siguiente:

```
Unset
register beto

connect beto
```

```
publish autores.txt texto descriptivo
list_content beto
quit
```

2. **Fichero Esperado** (test_files/expected/): Este fichero tiene el mismo nombre que el fichero de entrada pero con la extensión _expected. En él se encuentra la salida esperada que debería generar el servidor:

```
Unset
c> c> REGISTER OK
c> c> CONNECT OK
c> c> PUBLISH OK
c> c> LIST_CONTENT OK
       FILE0: <absolute_path>/autores.txt
c>
+++ FINISHED +++
```

3. **Fichero Output** (test_files/output/): Este fichero **contiene la salida real** generada por el cliente. Su nombre es el mismo que el fichero de entrada pero con la extensión .output

Básicamente, el script test.sh ejecuta todos y cada uno de los test especificados en la carpeta /test_files/input/, guarda su salida en un fichero .output y compara linea a linea esa salida con la salida esperada. En caso de que la salida sea idéntica, el test pasará y en caso contrario fallará. Si falla, se puede revisar la salida que ha generado en el fichero .output.

No obstante, es importante mencionar que hay casos en los que la salida esperada no tiene por qué ser igual a la salida real. Por ejemplo, cuando un usuario se conecta, su hilo de escucha se asigna a un puerto libre. Este puerto es completamente arbitrario, por lo que no se puede saber con certeza cuál va a ser y por consiguiente, no es posible generar un fichero _expected. Por ello, el script de test reemplaza los números de puerto por la palabra reservada **PORT**. Exactamente lo mismo pasa con la ruta de un fichero publicado, que al ser absoluta, depende totalmente del directorio desde el que se haya ejecutado el fichero de test. Por ello, todas las rutas se sustituyen por la palabra reservada **PATH.**

Las únicas pruebas distintas son las de GET_FILE y GET_MULTIFILE. En estas, primero se prepara un escenario con tres usuarios (A, B y C). Los usuarios A y B se registran y conectan al sistema para, posteriormente, publicar un fichero llamado temp.txt (que se genera en el script y consiste en un conocido poema: *Ozymandias*). Una vez se haya publicado, se prepara el escenario del usuario C, el

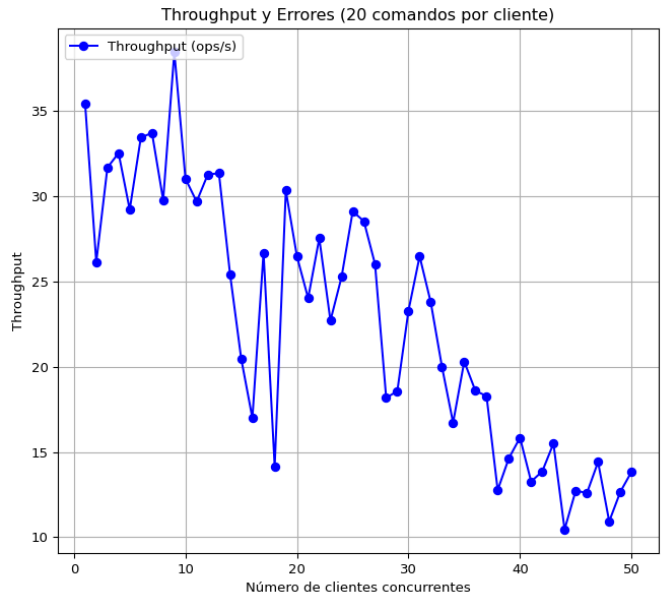
cual se registra, se conecta y en primer lugar, obtiene mediante GET_FILE el fichero temp.txt del usuario A. El fichero descargado se almacena en otro fichero llamado temp_download.txt. La prueba solo pasará si el fichero descargado es absolutamente idéntico al fichero temp.txt. De la misma manera se prueba GET_MULTIFILE. El usuario C hace una petición *get_multifile* al servidor y descarga, en otro fichero temp_multidownload.txt, el contenido de temp.txt. El fichero descargado se almacena en otro fichero llamado temp_download.txt. La prueba solo pasará si el fichero descargado es absolutamente idéntico al fichero temp.txt.

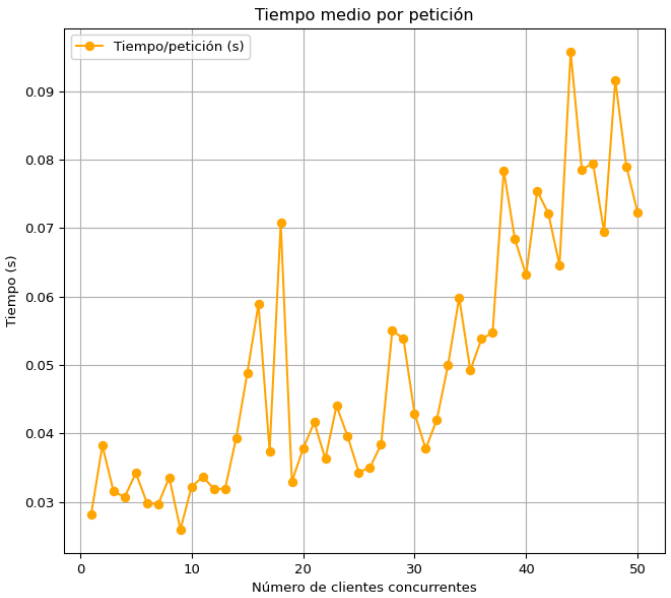
En total, se han realizado 25 test funcionales que prueban todos los posibles valores de retorno del servidor. Es importante mencionar que estos test, también prueban implícitamente el funcionamiento del logger (servicio rpc) y el *web_service*, ya que en caso de que estos dos componentes no estuvieran funcionando correctamente, la salida generada no sería igual a la salida esperada. Para probar esto, se recomienda al corrector ofuscar intencionadamente el fichero de test, comentando alguna línea en la que se inician estos servicios. (lineas 58 a 69 en el fichero test.sh).

De igual manera, es importante mencionar que hay valores de retorno que, debido a la solidez de la implementación, no es posible comprobar. Por ejemplo, en la función *PUBLISH* existe un mensaje de error en caso de si el usuario no está registrado y otro en caso de que el usuario no esté conectado. Esto es contraintuitivo, ya que el sistema está preparado para dar error en caso de si el usuario no está conectado (lo más restrictivo posible) y evidentemente, si un usuario no está registrado, tampoco va a estar conectado, por lo que únicamente se puede comprobar el valor de retorno que devuelve el programa en caso de que el cliente no esté conectado. Aún así, **todos los valores de retorno especificados en el enunciado se han incluido en la arquitectura del programa.**

Evaluación de Rendimiento

Para probar el correcto funcionamiento del programa en situaciones con mucho tráfico de datos, se ha elaborado un script en Python capaz de medir el rendimiento del sistema operando sobre él simultáneamente varios clientes al mismo tiempo. Después de un análisis exhaustivo, estos son los resultados ofrecidos por el servidor:





La gráfica de la izquierda representa el número medio de operaciones/segundo que ha podido realizar el servidor en función del número de clientes operando al mismo tiempo. Por ejemplo, Con 30 clientes

operando simultáneamente sobre el servidor, el throughput medio que éste ha podido ofrecer es de 23 operaciones por segundo aproximadamente. Es notoria la caída del throughput del servidor en función del crecimiento del número de programas.

Cabe destacar que éste experimento se realizó con todos los servicios en ejecución (logger, web service, servidor y cliente). En caso de que no hubiera estado activo el web service, se podría haber conseguido una menor latencia debido al tiempo que supone pedir al servicio web la fecha y hora. por otro lado, aumenta la velocidad el hecho de emplear sockets en lugar de RPC, ya que estos últimos generan más latencia.

En el experimento, también hay que mencionar que los 20 comandos de los clientes son elegidos de manera completamente aleatoria, pudiendo, por ejemplo, tratar de registrar un usuario que ya está en el sistema o intentar desconectar un usuario que no existe. De esta manera, estadísticamente se prueban todos los posibles escenarios que pueden coexistir en el servidor pero que también afecta al desempleo del experimento.

Contenido Extra

Función Get Multifile

Como contenido adicional, se ha implementado la función Get Multifile, que permite la descarga de un fichero **desde varios clientes simultáneamente**. Se ha tratado de seguir la misma filosofía que el resto de funciones en cuanto a mensajes de error por parte del servidor y protocolos de comunicación. De forma esquemática, la función está implementada de la siguiente manera:

1. El cliente crea un socket y se conecta con él al servidor, enviando la cadena de caracteres: “GET_MULTIFILE”. Posteriormente, envía otra cadena de caracteres con *timestamp* (obtenido del web service) y por último, el nombre de usuario del usuario que envía la petición.

2. El servidor responde con un código de operación (un Byte):
    a. Código de Operación 0: Éxito.
    b. Código de Operación 1: Ningún usuario actualmente conectado tiene el fichero que se ha especificado[2].
    c. Código de Operación 2: Cualquier otro error que pueda haber surgido en la parte del servidor.
3. En caso de que el código de operación sea 0, el cliente procede a recibir del servidor otro número (un Byte) que especifica el número de usuarios que tienen el fichero. Nótese, que al enviar un Byte, el número máximo de usuarios con un mismo fichero es como máximo 255 (2^8).
4. Por cada usuario con el fichero, el servidor procede a enviar la ip, el puerto y la ruta donde se encuentra el fichero en la máquina del usuario. Todos codificados en ASCII.

[2] Para que un fichero que puede estar en diferentes rutas (paths) se considere “el mismo” se ha considerado que un fichero es el mismo si tiene el mismo nombre. Es decir, el servidor considerará el fichero C:\User\Escritorio\foo.txt exactamente el mismo al fichero /home/user/Documents/foo.txt

5. Cuando el cliente ha recibido esta información, procede a crear un hilo por cada usuario que tiene el fichero deseado. Cada hilo, se encarga de crear un socket nuevo, conectarse a la ip y puerto del usuario al que ha sido asignado y envíar la siguiente información en orden:
    a. La cadena de caracteres “GET_MULTIFILE”.
    b. La ruta del fichero que desea obtener.
    c. Su “seeder_id”. Esto es, el identificador que distingue al hilo de los otros hilos que también están recibiendo parte del fichero.
    d. El número total de “seeders” (Hilos que están recibiendo el fichero de otros usuarios)
6. Una vez enviada la información, se procede a recibir un Byte de confirmación por parte del usuario (0 en caso de éxito y 1 en cualquier otro caso)
7. En caso de que el código de operación haya sido 0, el hilo procede a recibir por parte del cliente el fragmento del fichero que le corresponde. Es decir, en caso de que hubiera 2 clientes con un mismo fichero habría 2 seeders, por lo que el cliente interesado generaría 2 hilos. Cada uno de los hilos se conecta a un seeder distinto y le especifica mediante su seeder id el número de hilo que le corresponde. Es decir el primer cliente que envía el fichero recibiría un seeder id = 1 / 2 y el segundo cliente recibiría un seeder id 2 / 2. De esta manera, el primer cliente sabe que tiene que enviar el primer 50% del fichero y el segundo seeder tiene que enviar el 50% restante.
8. Una vez los hilos han recibido las diferentes partes del fichero de cada seeder, lo almacenan temporalmente en ficheros temporales en cuyo nombre se especifica el fragmento del fichero que es para que al ensamblarlo, se sepa el orden de ensamblado.
9. Finalmente, el fichero se ensambla con todos los ficheros temporales generados por los hilos y se procede a eliminar todos los ficheros temporales.

A pesar de que normalmente solo se dispone de una interfaz de red para recibir información externa, paralelizar la recepción de un fichero aumenta significativamente la velocidad de descarga, permitiendo realizar a cada cliente más operaciones por segundo y mejorando en general la eficiencia y experiencia de usuario de cada cliente.

No obstante, cabe destacar que si el fichero es pequeño, la carga computacional que conlleva la creación y gestión de hilos hace que salga más rentable descargar el fichero de un único usuario.

Recuperación de Errores

Se ha realizado la implementación de todos los componentes para que el cliente sea capaz de funcionar sin necesidad de que el web_service esté activo. De la misma manera, el servidor también puede funcionar sin necesidad de que el servicio RPC esté activo.

Con esta implementación, se permite depurar de manera mucho más sencilla y se gestiona de manera mucho más laxa las dependencias y programas en las que se basa el programa (paquetes de Python, rpcgen…)

Analizador de Paquetes de Red

Tal y como se comenta en el siguiente apartado, se ha realizado un vídeo que demuestra toda la conexión entre cliente y servidor analizando toda la comunicación entre los mismos mediante un capturador de paquetes (Wireshark):

https://youtu.be/L-rlU4EP5b0?si=_yo3e2jjMQy6FT2k

Conclusiones y Consideraciones Adicionales

Lagunas de Seguridad Importantes

Al no usar ningún tipo de algoritmo de encriptado entre cliente y servidor, más que “lagunas” de seguridad, se podría definir mejor como “océanos” de vulnerabilidades. Todo mensaje cliente servidor o cliente cliente, puede ser interceptado mediante un analizador de paquetes de red (como Wireshark) y al no estar encriptado, en la propia herramienta se puede discernir en texto plano toda comunicación entre cliente y servidor. Para mostrar este aspecto, se ha elaborado un vídeo demostrativo:

https://youtu.be/L-rlU4EP5b0

Además, es importante tener en cuenta que el hilo que gestiona la escucha de peticiones en la parte del cliente (el que se encarga de envíar ficheros) debería comprobar si el fichero que se ha solicitado para descargar, ha sido subido por el cliente. Es decir, técnicamente, **podrías pedir descargar un fichero a un cliente que NO HA SIDO PUBLICADO PREVIAMENTE EN EL SERVIDOR** debido a que no se realizar ninguna comprobación de seguridad.

Limitaciones en Red Local (UPnP)

Tal y como se ha especificado anteriormente, se ha querido probar el funcionamiento del programa en circunstancias que cubran todos los posibles casos de uso reales de una aplicación de este estilo. Uno de estas condiciones es muy real: ejecutar el cliente y el servidor en máquinas distintas.

Debido a que el servidor está escuchando siempre en el mismo puerto (que hay que introducir como argumento y que se debe conocer con anterioridad) es posible ejecutarlo en una máquina distinta fuera de la red local si se especifica en el router al que está conectado el servidor una regla de **port forwarding** redirigiendo todo el tráfico entrante en el puerto designado (en mi caso, el 4444) a la ip interna de la máquina que aloja el servidor dentro de la red local.

Esto funciona de maravilla para toda comunicación en el programa que consista en cliente - servidor. Sin embargo, esto no funciona cuando la comunicación es cliente-cliente. Ya que el hilo que escucha peticiones entrantes del cliente define un puerto de escucha aleatorio, por lo que si al router al que está conectado el cliente le llega una petición externa al puerto interno del cliente, no va a saber hacia dónde tiene que redirigir la información al no haber sido registrada en la tabla de encaminamiento del router.

Esto no sucede si ambos clientes se encuentran dentro de la misma red local, ya que la ip interna de cada uno es distinta, por lo que el router sí sabría hacia dónde redirigir la información. Lo que no pasa

cuando la petición viene desde fuera de la red local, ya que todo dispositivo de dentro de una misma red local tiene la misma ip pública/externa.

Como solución a esto, se podría implementar en el cliente un servicio que se conecte al router mediante el protocolo UPnP (Universal Plug and Play), que es capaz de realizar port forwarding sin necesidad de entrar en la configuración del router. Para esto, evidentemente hay que habilitar la opción de UPnP en el router y se debería introducir un módulo distinto en el código de Python del cliente.

A pesar de que esto se ha probado y ha funcionado sin problemas, el módulo no está disponible en Guernika, por lo que se ha decidido eliminarlo. Además, hacía que la velocidad del cliente disminuyera significativamente.

Como solución real lo que se debería hacer es que el cliente NO eligiera de forma aleatoria el puerto de escucha, si no que escogiera un puerto estandarizado para poder recibir peticiones entrantes de parte de otros clientes. Esta es la solución que se usa en sistemas torrent reales.

Conclusiones Generales

Esta práctica me ha ayudado a entender de verdad qué significa diseñar y trabajar con sistemas distribuidos. Hasta ahora muchos de los conceptos de la asignatura me resultaban algo abstractos, pero al tener que implementarlos y ver cómo se comportan en situaciones reales —con varios procesos, peticiones simultáneas y posibles errores— es cuando todo ha empezado a encajar.

Uno de los mayores aprendizajes ha sido darme cuenta de la complejidad que conlleva coordinar distintos componentes que se comunican entre sí, muchas veces de forma asincrónica y en paralelo. También he comprendido por qué este tipo de sistemas están en el corazón de muchas tecnologías punteras hoy en día: desde servicios web modernos hasta aplicaciones a gran escala que usamos todos los días.

En conclusión, esta asignatura (y más concretamente, esta práctica), su desarrollo y gestión cumplen todos los requisitos que idealicé cuando me metí en esta carrera y que ahora por fin puedo elaborar.