Universidad Carlos III de Madrid

Universidad Carlos III

Curso Sistemas Distribuidos 2024-25

Entrega Final
Curso 2024-25

Sistemas Distribuidos

Grado Ing Informática

Fecha: **11/05/2025** - ENTREGA: **FINAL**

GRUPO: **80**

Alumnos:

Hector Álvarez Marcos 100495794

Angela Elena Serrano Casas 100475053

Introducción

El presente documento contiene el desarrollo de la práctica final de la asignatura de Sistemas distribuidos. La práctica consiste en el diseño, implementación y desarrollo de un sistema peer-to-peer (P2P) para la distribución de ficheros. El sistema se divide en tres partes:

1. Un sistema P2P basado en sockets TCP.
2. Un servicio web para proporcionar la fecha y hora
3. Un servicio de llamadas a procedimientos remotos (RPC) para el registro de operaciones.

Para el correcto desarrollo de la práctica se aplicarán los conocimientos clave obtenidos a lo largo de la asignatura, como son la comunicación entre procesos concurrentes, todo ello en un entorno práctico que simula un escenario real.

El sistema desarrollado permite a los usuarios registrarse, conectarse, publicar y eliminar ficheros. También pueden transferirlos directamente entre clientes con el servidor coordinando las operaciones. La implementación de las funcionalidades se ve ampliada además con la integración de un servicio web y un servicio RPC. Esta memoria detalla los fundamentos teóricos, la arquitectura, la metodología, los retos enfrentados, y las conclusiones obtenidas.

Objetivos

La práctica persigue los siguientes objetivos:

- Comprender y aplicar los principios de comunicación en sistemas distribuidos mediante sockets TCP, servicios web, y RPC.
- Diseñar e implementar un sistema P2P que permita la gestión y transferencia de ficheros entre clientes.
- Desarrollar un servicio web en Python que proporcione la fecha y hora actual para incluir en las operaciones.
- Implementar un servicio RPC en C que registre las operaciones de los usuarios, justificando la interfaz definida.
- Evaluar el sistema bajo diferentes escenarios, garantizando robustez y correcto funcionamiento en entornos distribuidos.
- Documentar el proceso de desarrollo, incluyendo pruebas, retos, y soluciones adoptadas.

Fundamentos teóricos

En vistas de desarrollar una documentación del proyecto completa y fundamentada, destinamos el desarrollo del siguiente apartado a la descripción de los conceptos teóricos que contextualizan los

requerimientos y funcionalidades del sistema a implementar. Todos ellos, a su vez, basados en los conocimientos de la asignatura.

Sistemas Peer-to-Peer (P2P)

Un sistema P2P es un tipo de arquitectura distribuida donde los nodos actúan como clientes y servidores simultáneamente, compartiendo recursos directamente sin depender de un servidor central para el almacenamiento de datos. Los sistemas P2P se caracterizan por su escalabilidad y descentralización, ideales para aplicaciones como la distribución de ficheros.

Sockets TCP

Para la implementación de la comunicación entre entidades en la primera parte de la práctica se requiere el uso del protocolo TCP. Este implementa dicha funcionalidad por medio del uso de sockets TCP, que proporcionan un canal orientado a conexión, fiable y bidireccional. Los sockets TCP garantizan la entrega ordenada de datos, lo que es crucial para el protocolo de paso de mensajes definido en la práctica.

Servicios Web

Un servicio web es una aplicación que expone operaciones accesibles a través de protocolos estándar como HTTP. En la parte 2, se utiliza un servicio web en Python para devolver la fecha y hora en formato “hh:mm:ss dd/mm/aaaa”. Esto introduce el concepto de interoperabilidad entre componentes heterogéneos.

Llamadas a procedimientos remotos (RPC)

El modelo RPC, utilizado en la parte 3, permite invocar funciones en un servidor remoto como si fueran locales. ONC-RPC, implementado en C, utiliza un fichero de interfaz (.x) para definir las operaciones, que en este caso registran las acciones de los usuarios (nombre, operación, fecha/hora). La elección de ONC-RPC garantiza compatibilidad con sistemas UNIX/Linux.

Arquitectura del Sistema

Esta sección describe la arquitectura diseñada para la práctica, incluyendo los componentes principales y su interacción. El sistema se compone de los siguientes componentes, organizados en una arquitectura híbrida cliente-servidor y P2P:

- Servidor P2P: Un servidor concurrente en C que coordina las operaciones (registro, conexión, publicación, etc.) y mantiene la lista de usuarios y ficheros publicados en una base de datos SQLite.
- Cliente P2P: Un cliente multi hilo en Python que permite a los usuarios interactuar con el sistema y transferir ficheros directamente con otros clientes.

- Servicio Web: Un servicio SOAP desplegado localmente en cada cliente, que proporciona la fecha y hora actual.
- Servidor RPC: Un servidor ONC-RPC en C que registra las operaciones de los usuarios, accesible desde el servidor P2P. Haciendo uso de la variable de entorno LOG_RPC_IP. Este servicio actúa de servidor

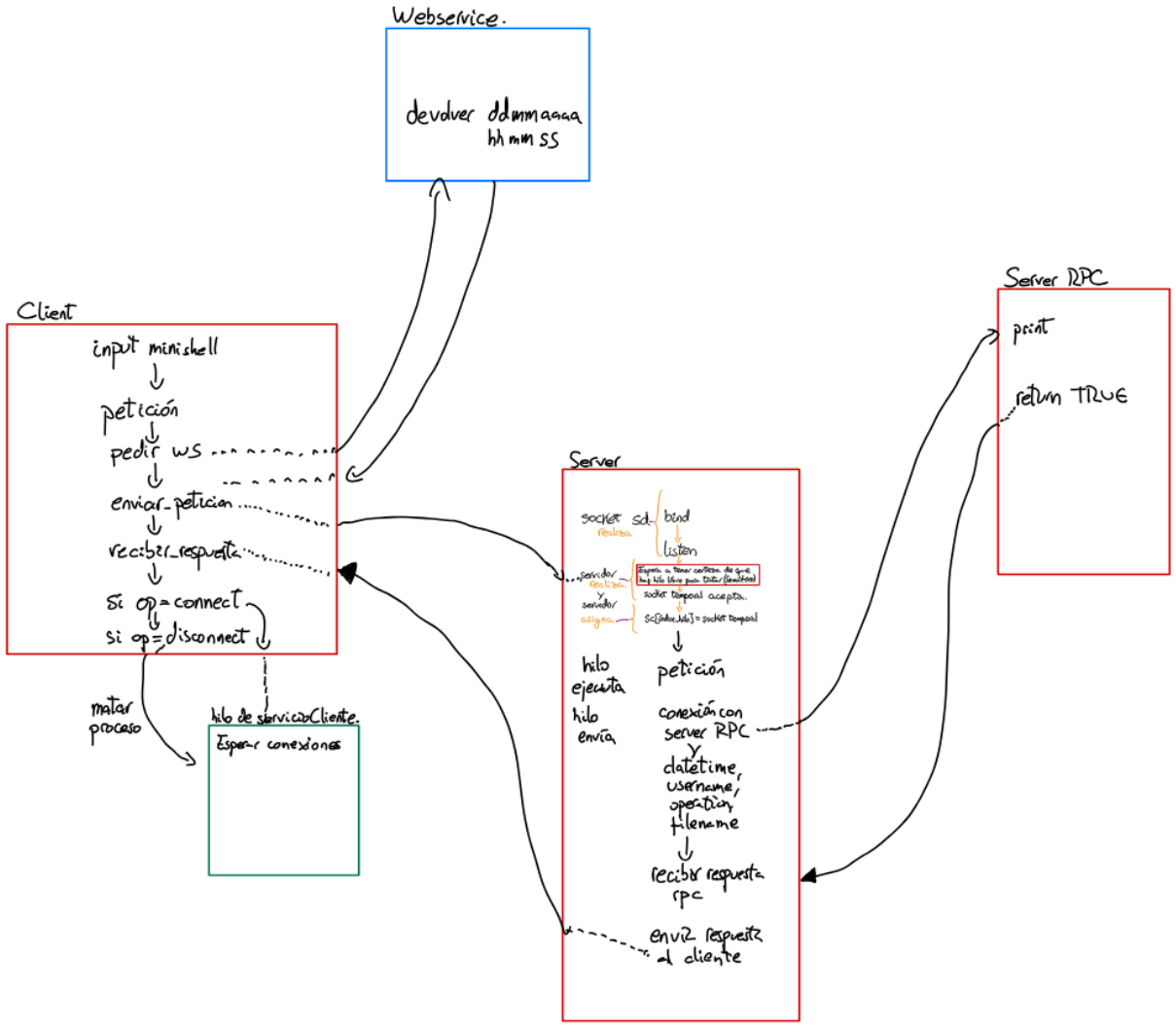
Protocolos de comunicación

A continuación, se definen los protocolos de comunicación a implementar en las distintas entidades que interactúan con el sistema.

1. Para la primera parte de la práctica se utilizan sockets TCP con un protocolo de paso de mensajes basado en cadenas de caracteres. Cada operación (REGISTER, PUBLISH, etc.) sigue un formato específico, con códigos de error definidos.
2. Para la comunicación con el servicio web se utiliza SOAP para devolver la fecha/hora en formato texto.
3. Finalmente para la implementación con ONC-RPC se utiliza un protocolo basado en TCP y UDP, con una interfaz definida en un fichero .x.

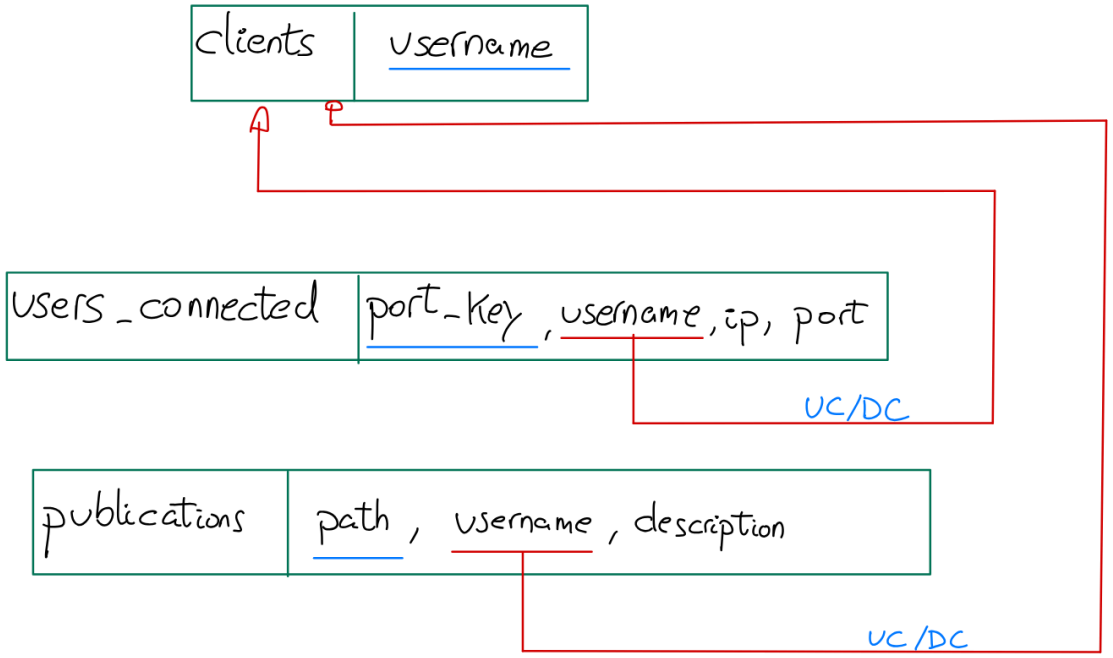
Diagrama de la comunicación

En el siguiente diagrama se puede ver cómo se produce la comunicación entre todos los servicios existentes en el sistema



Abstracción del sistema de almacenamiento

Para el almacenamiento hemos optado por un sistema de guardado en base de datos SQL, el cuál es persistente al cerrado de los procesos gestores del servidor, lo cual garantiza continuidad y completitud de los datos a lo largo del tiempo. el sistema de guardado se puede explicar mediante el siguiente diagrama relacional:



En el se puede ver como se ha optado por una generación de 3 tablas, clients, users_connected y publications. Las 2 últimas están directamente relacionadas con la primera mediante un UC/DC, el cuál implica que si se elimina la referencia al usuario en la tabla clients, se eliminaran todas las filas en las otras tablas.

Justificación de la interfaz RPC

Simplicidad: La interfaz RPC define una operación que acepta 4 parámetros de tipo string(username, operation, filename, datetime) para registrar múltiples acciones (REGISTER, PUBLISH, …), indicando el usuario y la hora a la que se realizaron. Se induce una sola operación, reduciendo la complejidad al centralizar la lógica en el servidor P2P.

Flexibilidad: El uso de cadenas para especificar la operación y sus parámetros permite manejar diferentes tipos de acciones sin modificar la interfaz, adaptándose a los requisitos del protocolo de la parte 1.

Eficiencia: ONC-RPC es adecuado para sistemas UNIX/Linux, y el formato de cadenas es compatible con el protocolo basado en \0 usado en la comunicación cliente-servidor.

Descripción del código

Esta sección describe los componentes implementados hasta la fecha, incluyendo el cliente P2P, el servicio web, el servidor P2P y la conexión con el servidor RPC.

Cliente P2P (client.py)

El cliente P2P, implementado en Python, está encapsulado en la clase client, que proporciona métodos estáticos para interactuar con el servidor P2P y otros clientes. La implementación utiliza sockets TCP para la comunicación con el servidor y entre clientes, siguiendo el protocolo de paso de mensajes definido en la práctica. Los principales métodos y características son:

- Atributos principales:
    - _server y _port: Almacenan la IP y puerto del servidor, configurados mediante argumentos de línea de comandos.
    - _user: Nombre del usuario conectado actualmente.
    - _thread y _stop_event: Gestionan un hilo daemon que escucha peticiones de otros clientes para transferencias de ficheros.
- Métodos de comunicación:
    - recv_until_null: Recibe datos desde un socket hasta encontrar un byte nulo (\0), decodificando cadenas UTF-8 o retornando enteros según corresponda.
    - send_op_and_user: Envía la operación y el nombre de usuario al servidor, siguiendo el formato del protocolo.
    - rcv_info_byte: Interpreta el byte de respuesta del servidor, mapeándolo a códigos de error definidos en la enumeración RC (e.g., OK, REG_USE_ERR). En estos códigos de error se especifican todos los indicados en el enunciado de la práctica
- Operaciones principales:
    - register y unregister: Gestionan el registro y eliminación de usuarios en el servidor.
    - connect y disconnect: Establecen y cierran conexiones, iniciando un hilo para escuchar peticiones de transferencia de ficheros en un puerto libre.
    - publish y delete: Publican y eliminan ficheros, enviando el nombre del fichero y su descripción al servidor.
    - listusers y listcontent: Obtienen la lista de usuarios conectados y el contenido publicado por un usuario específico, mostrando la información recibida.
    - getfile: Solicita un fichero a otro cliente, utilizando la información de IP y puerto obtenida mediante listusers.
    - Connect: Establece una conexión, generando un hilo con un socket de escucha de peticiones. Esto lo convierte a su vez en servidor para otros clientes que quieran obtener su archivo. Esto se realiza mediante la función hear_petitions.
- Interfaz de comandos (shell): Proporciona una consola interactiva que interpreta comandos como REGISTER, PUBLISH, GET_FILE, etc., validando la sintaxis y ejecutando las operaciones correspondientes.
- Gestión de concurrencia: El método hear_petitions ejecuta un hilo que escucha en un puerto libre para responder a peticiones GET_FILE de otros clientes, enviando el contenido del fichero solicitado o un código de error si no existe.
- El cliente maneja errores robustamente, verificando la conexión del usuario y la validez de los parámetros antes de enviar peticiones. La enumeración RC centraliza los códigos de error, facilitando la gestión de respuestas del servidor.

Aclaraciones sobre funcionamiento del cliente

Para aclarar posibles zonas grises, o errores y fallos que hemos considerado en el código aparte de lo estrictamente solicitado, vamos a explicar cómo hemos resuelto estos puntos y el por qué los consideramos así.

Primeramente desde un proceso client.py, solo puede conectarse un usuario a la vez. Nosotros para garantizar esto, guardamos dentro de una variable de clase _user el nombre del usuario conectado siempre y cuando el servidor devuelva que la conexión ha sido exitosa. En el caso de que tengas 2 usuarios registrados, “a” y “b” y desde un proceso conectes “a”, si desde ese mismo proceso deseas conectar “b” puedes hacerlo sin necesidad de desconectar primero a “a”, ya se encarga la propia función connect de llamar a la desconexión de “a” para así poder desconectar a “b”.

Esto viene a colación porque el único lugar en el que puede producirse el fallo connect fail, user already connected, es que a través de un proceso client.py esté “a” conectado, y desde el otro proceso client.py , se intente conectar a “a”.

También consideramos que desde un proceso distinto, no puedes desconectar a un cliente que no viene de tu proceso, al igual que no puedes borrar contenido publicado por otros usuarios.

También para el tema de las rutas, se admiten en todos los casos tanto rutas relativas como absolutas. Salvo para dar la ruta de origen en la función get_file. El propio cliente buscará, en los casos admisibles, la ruta absoluta de esa ruta relativa publicada. En cualquier caso, para el publish, no se permitirá enviar al servidor la ruta de un archivo que no exista.

Servicio Web (ws-time.py)

- El servicio web, implementado en Python, utiliza el framework Spyne para ofrecer un servicio SOAP que proporciona la fecha y hora actual. Sus características principales son:
    - Estructura: Define una clase GetTime con un único método get_time.
    - Funcionalidad: El método get_time retorna la fecha y hora en formato hh:mm:ss dd/mm/aaaa, utilizando las funciones time.localtime y time.strftime.
    - Protocolo: Utiliza SOAP 1.1 para la entrada y salida, con validación XML mediante lxml.
    - Despliegue: El servicio se ejecuta en 127.0.0.1:8000 utilizando wsgiref.simple_server, accesible localmente por el cliente P2P.

Servidor P2P

El servidor P2P, es un servidor concurrente multihilo que coordina las operaciones de los clientes y gestiona la información en una base de datos SQLite.

El enunciado imponía el uso de sockets TCP, una base de datos SQLite con tablas clients, users_connected, y publications, y un protocolo de mensajes terminados en \0. Se decidió implementar el servidor en C para un mejor control de recursos, utilizando un thread pool de 25 hilos para gestionar la concurrencia. La sincronización se logró con mutex y semáforos, y se implementó manejo de endianness para garantizar compatibilidad en la comunicación.

La inclusión de este servicio de threads reutilizables garantiza la no escalabilidad en memoria por generación de hilos detached la cual puede acabar en colapso del sistema debido a escasez de recursos

de la memoria otorgada para el proceso. Este sistema, garantiza, que, para alto flujo de peticiones entrantes, se gestione de forma rápida y eficiente además de controlar la gestión de memoria.

A continuación, se describen sus componentes principales:

- Arquitectura:
    - Inicialización: Crea un socket TCP, realiza bind y listen en el puerto especificado, y configura una base de datos SQLite con las tres tablas: clients (usuarios registrados), users_connected (usuarios conectados con IP y puerto), y publications (ficheros publicados con descripciones).
    - Gestión de hilos: Utiliza un thread pool de tamaño fijo, gestionado mediante semáforos y mutex para asignar conexiones a hilos libres. Cada hilo ejecuta process_request, que procesa una petición y libera el hilo al finalizar.
    - Comunicación: Acepta conexiones mediante accept y delega el manejo de cada conexión a un hilo, pasando la IP del cliente y el descriptor del socket.
- Base de Datos (database_control.c):
    - Estructura: Utiliza SQLite para gestionar las tres tablas mencionadas previamente. Hace uso de claves foráneas y restricciones de integridad. Las operaciones están protegidas por un mutex para garantizar una concurrencia segura.
    - La base de datos se inicializa con un nombre dinámico basado en el usuario del sistema (/tmp/database<user>-.db)
    - Operaciones: Implementa las funciones requeridas en el enunciado como register_user, connect_client, publish, etc., que ejecutan consultas SQL para insertar, eliminar o consultar datos, almacenando los datos asociados a cada funcionalidad del sistema.
    - Integridad: Habilita claves foráneas para mantener la consistencia entre tablas.
- Comunicación (message_control.c):
    - Recepción: Se leen las cadenas terminadas en \0 desde el socket, deserializándolas en una estructura request que contiene la operación, usuario, puerto, ruta, descripción, etc.
    - Envío: La función send_message envía un byte con el código de respuesta, mientras que send_message_query envía informacion relativa a la consulta realizada y sean listas de usuarios o contenido publicado por estos.
    - Endianness: Maneja diferencias entre little y big endian para garantizar compatibilidad en la transmisión de datos.
- Consultas SQL (sql_recall.c):
    - Callbacks: Define funciones de callback que procesan filas devueltas por consultas SQL y almacenan resultados en estructuras como receive_sql o request_query_client
    - Uso: Estas funciones son utilizadas por list_users y list_content para devolver listas de usuarios conectados o contenido publicado.

Servidor RPC

Hemos añadido concurrencia en el servidor RPC, ya que compilando los stubs con -M nos permite generar funciones MT safe, pero la concurrencia tiene que ser implementada por nosotros. Tenemos un sistema que utiliza una pool de hilos(50) que se encargan de esperar a que las funciones

implementadas por el servicio RPC añadan las peticiones a una cola. Estos hilos se encargarán de ir extrayendo elementos de la cola y realizando la funcionalidad, que en este caso es imprimir por terminal.

Compilación y ejecución

En este apartado, se explicará como se ha compilado este proyecto, así como las normas y reglas que se tienen que seguir para realizar la correcta ejecución del sistema.

Entorno de ejecución

El sistema fue desarrollado y probado en guernika-gpu.lab.inf.uc3m.es, utilizando compiladores y versiones de Python compatibles con el entorno de las aulas informáticas. No se requiere software adicional más allá de las bibliotecas estándar y ONC-RPC.

Compilación

Servidor P2P y Servidor RPC: Compilar usando el Makefile proporcionado: “make”

Para la compilación de este sistema se ha generado un fichero Makefile, el cuál es el encargado de compilar todos los ficheros relacionados con el lenguaje C. Este fichero, se encarga de enlazar para el ejecutable servidor, todos los objetos que garantizan el correcto funcionamiento del sistema. También se encarga de generar y compilar los stubs del sistema RPC y la compilación del servidor RPC. Mediante la regla “make clean” se permite el borrado de la base de datos, permitiendo “reiniciar” el sistema de almacenamiento, por si se hubiera producido algún problema o error inesperado.

Ejecución

1. Iniciar el servicio web en la máquina local del cliente: (Desde el directorio raíz)

   Como el puerto que se usaba por defecto puede llegar a estar ocupado, hemos implementado que desde la terminal cuando se ejecute el fichero se indique el puerto donde desea lanzar el servicio. Por defecto tenemos un shell script “init_webservice.sh que ejecutara el servicio en el puerto 4567, pero si no el funcionamiento es el siguiente

   python3 ./src/client/ws-time.py -p <port>

   &

   python3 -mzeep http://localhost:<port>/?wsdl

   Para facilitar la conexión hemos generado un fichero init_webservice.sh que ejecuta ambos comandos con el puerto 4567, que ha sido el que por defecto se usa en nuestro código.

2. Iniciar el servidor RPC:

   ./servidor_rpc

3. Iniciar el servidor P2P, especificando la variable de entorno LOG_RPC_IP:

export LOG_RPC_IP=<IP_del_servidor_RPC>

En guernika solo funciona si se especifica como “localhost”.

./servidor -p <puerto>

4. Iniciar el cliente P2P: (Desde el directorio raíz)

   python3 ./src/client/client.py -s <IP_servidor> -p <puerto servidor> -ws <Port ws>

   Para el código del cliente hemos añadido la flag -ws, que indica el puerto desde el cual se ha ejecutado el ws. En caso de no poner nada, considerará el puerto 4567, que es el default que hemos usado para el levantamiento del servicio

   Cabe destacar que si se quieren usar rutas absolutas en el publish, el sistema buscará la ruta absoluta desde el directorio donde se encuentra el proceso en el momento de ejecución. Si se ejecuta el comando indicado en el punto 4, las rutas absolutas se consideran desde el directorio raíz.

Ejecutables auxiliares

Pese a ir especificando a lo largo de la memoria todos los nuevos comandos incluidos en el sistema, vamos a realizar una recopilación para que quede claro para qué sirve cada uno.

Directorio raíz:

stress_test.sh: shell script pensado para probar el comportamiento paralelizable del servicio

init_webservice.sh: shell script pensado para facilitar el inicio del servicio web solicitado. Por defecto se abrirá el servicio en el puerto 4567.

delete_database.sh: shell script generado para poder realizar el borrado del archivo contenedor de la base de datos del sistema.

Directorio src/tests:

register_user.py: fichero de python generado para la prueba de estrés concurrente

run_all.sh: shell script pensado para realizar las pruebas de funcionalidades avanzadas.

IMPORTANTE QUE PARA EL CORRECTO FUNCIONAMIENTO DEL SH SE EJECUTE DESDE EL DIRECTORIO SRC/TESTS

Batería de pruebas y resultados

Para probar nuestro código hemos dividido las pruebas en 2 partes, pruebas de estrés y pruebas de funcionalidades avanzadas.

Prueba de estrés

Esta prueba se ha realizado para probar y garantizar la escalabilidad del sistema y el comportamiento concurrente de sendos servicios considerables como servidor, que son el servidor del sistema, el servidor RPC y el webservice. Ambos deben garantizar el acceso concurrente, para que la usabilidad del sistema sea máximo. Para ello hemos generado un archivo .py en el directorio src/tests/ el cual implementa las funcionalidades register y connect para un random UUID(Universal Unique IDentifier) que actúa de username, para garantizar la correcta inserción en base de datos. La función connect en este caso, utiliza IP localhost y puerto un random.randint(1025, 65536). Como ya hemos indicado anteriormente, el identificador único de los clientes conectados en la base de datos es la concatenación de la IP_PORT del cliente, para garantizar que un mismo proceso no puede usarse en el mismo puerto del mismo sistema. Como en este caso no estamos generando hilos de escucha del cliente, puede producirse fallo por la colisión de puertos, aunque puede ser pura casualidad estadística.

Este archivo .py ha sido usado en un stress_test.sh, el cual encapsula sendas llamadas concurrentes al fichero .py, que ya de por sí realiza 10 registros/conexiones. Esto garantiza el correcto funcionamiento concurrente de ambos servicios, ya que usa los 3 “servidores” con comportamientos paralelizables.

Pruebas de funcionalidades avanzadas

Para la batería de funcionalidades avanzadas, nos hemos generado un shell script, que se encarga de probar tanto los casos válidos, como todos los casos de error de la aplicación capturando la salida en un fichero guardado en el directorio temporal temp/ y comparandola con la salida esperada ubicada en el directorio expected/.

A lo largo de las 26 pruebas realizadas, se verifican todos los casos de error y posibles puntos conflictivos de la aplicación y **es importante tener la base de datos limpia al momento de realizar la prueba**. Para ello bastará, con el servidor desconectado, con ejecutar el fichero delete_database.sh ubicado en el directorio raíz para borrar el archivo contenedor de la base de datos, y posteriormente levantar el servicio.

Este fichero sh está pensado con el servidor usando el puerto 3333 y el ws el puerto 4567 pero se puede cambiar según lo que se use mediante las variables port y wsport ubicados en la parte superior del ejecutable.

Para la batería de pruebas hay 2 funciones que pueden no dar success como resultado. Únicamente debido, y se verá, en el test 21(list_content_ok)la ruta guardada dentro del directorio expected es la que se realizó cuando como usuarios nosotros ejecutamos el fichero, por lo que la ruta no se corresponderá con la tuya. Otro que quizás falle sea el 19(list_user_ok) ya que almacena una dirección ip que quizás no sea la ejecutada, aunque ejecutándose en guernica-gpu no creo que haya problema.

Retos y soluciones

El mayor reto de esta práctica vino de la complejidad de desarrollar todo el esquema y la arquitectura de comunicaciones. Afortunadamente, ya conocíamos el comportamiento del lenguaje c con sockets y las llamadas a procedimientos de la base de datos mediante sqlite3, lo que ayudó al desarrollo óptimo del proceso del servidor. Esto facilitó el buen desarrollo del cliente en python, ya que nos permitió ocupar más tiempo en generar, tanto en cliente como en servidor, un código limpio, legible y fácilmente comprensible para cualquier usuario, lo que garantiza una calidad a la hora tanto de la corrección como de posibles mejoras a futuro. Otro de los retos fue descubrir de forma exhaustiva cómo realizar correctamente la concurrencia del servidor RPC, ya que al principio no comprendíamos que solo con el generar stubs MT safe no era suficiente, sino que de alguna forma teníamos que implementar un servidor concurrente usando esas funciones generadas.

La realización de una batería de pruebas usando bash también ha sido tedioso de realizar, pero consideramos que era necesario para verificar el correcto funcionamiento de nuestro código

Conclusiones

La práctica permitió consolidar conocimientos sobre sistemas distribuidos, aplicando

técnicas de comunicación (sockets TCP, HTTP, RPC) en un sistema P2P funcional.

La implementación de los tres componentes demostró la importancia de un diseño

modular y una gestión eficiente de la concurrencia. Los retos enfrentados, como la gestión concurrente de todos los procesos paralelizables, supuso un gran reto, así como la comunicación entre todos los usuarios del sistema, pero el proceso de realización de estos enriqueció el aprendizaje. Como mejora futura, se valoraría positivamente la inserción en alguna funcionalidad, a modo de básico, alguna implementación mediante el lenguaje rust, ya que es un servicio muy demandado últimamente, y que estaría muy interesante poder aprender este lenguaje tan solicitado hoy en día aplicándolo en un enfoque práctico en algún sistema real.

Desde un punto de vista personal, el desarrollo de está práctica supone un buen cierre de la asignatura, recapitulando los puntos más importantes de esta y aplicándolos en un contexto que podría suponer perfectamente un caso real. Consideramos que la práctica fue desafiante y a la vez gratificante y creemos que es supone una gran ayuda para los estudiantes puesto que consideramos que el desarrollo de esto ayuda a afianzar los conocimientos teóricos de la asignatura