Universidad Carlos III de Madrid

Sistemas Distribuidos

Documentación Práctica Final

Javier Martín Pizarro - 100495861

Alberto Pascau Sáez - 100495775

Repositorio de github utilizado: link
El repositorio es público, y de libre uso. Se ha comentado para que el lector pueda comprender de manera intuitiva el código y su funcionamiento.

1. Estructura del documento

Este documento describe de manera coherente, simple, intuitiva y resumida el proyecto final de sistemas distribuidos: un sistema *peer-to-peer* que conecta clientes con un único servidor.

Para ello, se puede definir la estructura de este proyecto en tres partes:

1. La primera parte **p1**. Incorpora la parte más importante de la práctica.
2. La segunda parte **Servicio Web**. Incorpora un servicio web que permite obtener la hora exacta en dicho momento y se procesa en el servidor.
3. La tercera parte **RPC**. Un sistema RPC, en vez de basarse únicamente en *sockets*.

Existe una sola carpeta de código, con todas las prácticas combinadas en ella. Es por ello que no se puede ver una evolución gradual del proyecto, sino la versión final.

Por último, el documento describe los tests usados para comprobar la eficacia y una breve conclusión que busca resumir las prácticas usadas.

2. Cliente

La clase `P2PClient` encapsula toda la lógica de cliente en Python para la aplicación P2P.

En el constructor (`__init__`) se almacenan las direcciones del servidor central (`server_host`, `server_port`) y del servicio de fecha/hora (`ws_host`, `ws_port`), además de inicializar el nombre de usuario, el socket de escucha y el evento de cierre.

`_do_server_op` unifica la comunicación con el servidor central: abre un socket, envía la operación y sus argumentos (incluyendo el timestamp, véase la sección del servicio web), recibe el código de respuesta y, en caso de operaciones de listado, recopila los datos devueltos.

Los métodos de alto nivel son:

- `register`/`unregister`: registran o eliminan un usuario en el servidor.
- `connect`/`disconnect`: establecen o cierran la conexión activa del cliente, manejando internamente un hilo de escucha para peticiones P2P.
- `publish`/`delete`: publican o eliminan contenido asociado al usuario en el servidor.
- `list_users`/`list_content`: solicitan al servidor los usuarios activos o los archivos publicados por un remoto, imprimiendo el resultado.

- `get_file`: tras obtener IP y puerto de un par con `LIST USERS`, abre conexión directa y descarga el archivo remoto en bloques.

La gestión del intercambio P2P recae en el hilo listener (`_listener_thread`) y su manejador (`_handle_peer`), que atiende operaciones `"GET FILE"` y envía el contenido en trozos de hasta 4 KiB. El método privado `_stop_listener` detiene esta escucha de forma ordenada.

3. Servidor

El servidor en C gestiona las peticiones de los clientes y despacha cada operación en un hilo independiente.

En la cabecera se incluyen utilidades (`server_utils.h`, `services.h`, `utils.h`, `limits.h`, `timestamp.h`) que proporcionan funciones de validación, registro de operaciones y generación de marcas de tiempo.

`handle_client`

- Recibe, por parámetro, el descriptor de socket y la dirección del cliente.
- `recv_string` obtiene primero la operación (`op`) y luego el `timestamp`, comprobando su longitud (esto es importante para el servicio web).
- Se protege el acceso compartido a la lista de usuarios con un `pthread_mutex_t` (`users_mutex`).
- Lee el nombre de usuario y, según `op`, llama a la función correspondiente de la capa de servicios (`register_user`, `unregister_user`, `connect_user`, `disconnect_user`, `publish`, `delete_s`, `list_users`, `list_content`).
- Para operaciones de listado, envía también los datos resultantes al cliente antes de liberar el mutex.
- Envía al cliente un byte con el código de estado y cierra el socket.

`controlC`

Captura la señal `SIGINT` (Ctrl+C) para cerrar el socket global (`sock`) de forma ordenada y finalizar el servidor con un mensaje.

`main`

- Configura el manejador de `SIGINT` con `signal(SIGINT, controlC)`.
- Comprueba que reciba un único parámetro (puerto) y crea el socket TCP (`socket()`, `bind()`, `listen()`).
- Entra en un bucle infinito de aceptación de conexiones (`accept()`), mostrando trazas con `printi`.
- Para cada cliente, reserva memoria para el descriptor y la dirección, lanza un hilo con `pthread_create(handle_client)` y lo libera automáticamente con `pthread_detach`.
- Cuando termina el bucle (al cerrar el socket), finaliza la aplicación.

4. Servicio Web

En la clase cliente, se añade un método privado que es capaz de obtener la hora del cliente. El método privado `_get_timestamp` consulta el servicio web de fecha/hora y devuelve una cadena con el formato `DD/MM/YYYY HH:MM:SS`, o una cadena vacía si falla.

El propio servicio web también se encuentra escrito en Python. Sigue la estructura estudiada en clase. Por defecto, se levanta en *localhost* en el puerto 5000. El servicio web es invocado a través del método privado y obtenemos el formato de la hora, que el cliente envía en todas de sus peticiones.

En el lado del servidor, el propio servidor sigue el protocolo e imprime los contenidos de la fecha en su terminal cuando es necesario.

5. RPC

El fichero de definición `rpc.x` (véase src/rpc/includes/rpc.x) establece la interfaz RPC principal:

- `program TIMESTAMP_PROG = 495775`: identificador único del programa en el registro RPC.
- `version TIMESTAMP_VERS = 1`: versión de la API RPC.
- Procedimiento `IMPRIMIR_OPERACION = 1`:
  - Parámetros: `string nombre_usuario`, `string operacion`, `string nombre_fichero`, `string timestamp`.
  - Retorno: `int` que indica el resultado de la operación remota.

Con el comando

```
rpcgen -MNa rpc.x
```

se generan automáticamente los ficheros de stubs y XDR: `rpc.h`, `rpc_clnt.c`, `rpc_svc.c` y `rpc_xdr.c`.

En el servidor:

- `rpc_svc.c` define el esqueleto que despacha las llamadas a `imprimir_operacion_1_svc()`, donde se deserializan los parámetros y se invoca la función de servicio `imprimirOperacion()`.
- La función `main` RPC registra el programa y versión con los transportes UDP y TCP (`svc_udp_create`, `svc_tcp_create`), asocia `TIMESTAMP_PROG/TIMESTAMP_VERS` al despachador y lanza `svc_run()` para atender peticiones entrantes.

Así, el cliente RPC (usando `rpc_clnt.c`) puede invocar `IMPRIMIR_OPERACION` de modo transparente, y el servidor procesa cada llamada en arquitectura evento-basada.

6. Diseño del protocolo

Toda la comunicación entre clientes y servidor (y entre pares) se realiza sobre TCP/IP. El protocolo de aplicación es un sencillo protocolo orientado a mensajes, basado en cadenas de texto y códigos de estado:

1. El cliente abre una conexión TCP al servidor central en `server_host:server_port`.
2. Envía la operación (por ejemplo, `REGISTER`) usando `send_string`.
3. Envía a continuación el `timestamp` y luego cada argumento (usuario, puerto, ruta de fichero, descripción, etc.) como cadenas.
4. El servidor procesa la petición, devuelve un único byte con el *código de estado*.
5. Para `LIST_USERS` y `LIST_CONTENT`, si el código de estado es 0 (éxito), el servidor envía:
   - Un entero `n` (como cadena) con el número de elementos.
   - A continuación `n` cadenas con los datos (usuarios — nombre, IP y puerto — o nombres de fichero).
6. El cliente cierra la conexión TCP.

Para la transferencia directa de archivos entre pares (P2P):

- Cada cliente abre un socket TCP en un puerto efímero y lo anuncia al servidor con `CONNECT`.
- El cliente que quiere descargar un fichero se conecta al par:
  1. Envía `"GET FILE"` y la ruta del fichero remoto.
  2. El par responde con un byte de estado:
     - `0`: fichero encontrado → envía tamaño (como cadena) y luego los datos en bloques.
     - `1`: fichero no existe.
     - `2`: error.
- Tras la transferencia, ambos cierran la conexión TCP.

Transporte: TCP/IP.

RCPResponse actúa como el equivalente de Response en la práctica de sockets, donde guardamos los valores necesarios (pueden ser NULL) y los devolvemos al proxy/cliente para que pueda ser procesados por su parte.

7. Tests

Los tests definidos en `tests_rpcServer()` son los siguientes:

`test_default` Verifica que la función `imprimirOperacion()` devuelva el código `0` (éxito) para cada una de las operaciones válidas: `CONNECT`, `DISCONNECT`, `REGISTER`, `UNREGISTER`, `PUBLISH`, `DELETE` y `LIST_CONTENT`.

`test_operacion_incorrecta` Comprueba que `imprimirOperacion()` devuelva el código `1` (error) al recibir una operación no reconocida (`"BAD_OP"`).

Hay ciertos tests que requieren de una conexión por parte de un cliente. Para esos no se han hecho tests automatizados, pero se propone una script .sh que cree un cliente en Python y lo lance al servidor, comparando los resultados esperados.

8. Conclusión

En este trabajo se ha diseñado e implementado un sistema P2P completo, con un cliente en Python que gestiona registro, conexión, publicación y transferencia de archivos, y un servidor en C que atiende de forma concurrente las peticiones de los clientes. La comunicación se basa en un protocolo simple sobre TCP/IP, con operaciones textuales y

códigos de estado, y se complementa con una capa RPC para el registro de operaciones mediante `rpcgen`.

La arquitectura modular—que separa claramente lógica de red, servicios y utilidades de validación—facilita la extensión y el mantenimiento del código. Además, la batería de tests automatizados para las funciones (`imprimirOperacion`) garantiza la fiabilidad de la capa RPC y sienta las bases para futuros ensayos sobre el resto de funcionalidades.

En conjunto, el sistema demuestra un flujo coordinado entre componentes centralizados y enlaces P2P directos, ofreciendo un punto de partida sólido para ampliar características como cifrado de las transferencias, gestión de errores avanzada o interfaces gráficas de usuario.