uc3m | Universidad Carlos III de Madrid

Sistemas Distribuidos

Ejercicio 2 - Sockets

Marzo 2025

Ruben Castro Pruneda
100454441@alumnos.uc3m.es
Doble Grado Informática y ADE

Alberto Molina Felipe
100454278@alumnos.uc3m.es
Doble Grado Informática y ADE

Diseño

Los cambios sobre el anterior ejercicio evaluable se reducen a la lógica del proxy y servidor que se ocupan de la comunicación. Lo que antes era mandar y recibir por la cola de mensajes de `POSIX` ahora es serializar y deserializar el mensaje por un socket.

Un hilo principal se encarga de leer continuamente las solicitudes entrantes a través del socket, y por cada solicitud recibida se crea un nuevo hilo que construye y envía la respuesta utilizando el mismo socket. Para coordinar el acceso concurrente a este recurso compartido, se utilizan mutex, asegurando comunicación sin interferencias.

Se han añadido funciones de pack y unpack tanto para las solicitudes como para las respuestas. Es importante destacar que las funciones de serialización y deserialización deben estar coordinadas, ya que el contenido serializado depende del orden y del tipo de los campos incluidos. Por esta razón, el orden en que se serializan los campos debe coincidir exactamente.

Comunicación

La comunicación entre el proxy y el servidor se realiza mediante el envío de estructuras `message_t` a través de sockets, utilizando funciones de serialización (`pack`) y deserialización (`unpack`).

En las **solicitudes**, siempre se envían el comando, el identificador del hilo emisor y el tipo. Si el comando no es `DESTROY`, también se incluye la clave. En los comandos `SET` o `MODIFY`, se añaden además los campos adicionales.

En las **respuestas**, siempre se devuelven el comando, el resultado de la operación, el identificador del hilo emisor y el tipo. Si se trata de un comando `GET` exitoso (es decir, con resultado igual a 0), se incluyen también la clave y los valores de la tupla.

Es fundamental que las funciones de serialización y deserialización estén perfectamente sincronizadas: los datos deben enviarse y leerse en el mismo orden. En particular, es imprescindible leer primero el comando, ya que determina qué otros campos deben leerse a continuación.

Compilación

Como indica el enunciado se compila el proxy como una biblioteca compartida y se generan dos binarios `bin/cliente` y `bin/servidor` con el comando `make`, ambos aceptan un flag para la verbosidad de los prints por pantalla.

Para ejecutar el servidor es necesario incluir el puerto como argumento y funcionará en localhost.

```
$ ./bin/servidor <puerto>
```

Para el cliente es hace falta tener las variables de entorno `IP_TUPLAS` y `PORT_TUPLAS` definidas.

```
$ env IP_TUPLAS=localhost PORT_TUPLAS=<port> ./bin/cliente
```

Tests

Los tests son los mismos que en el ejercicio anterior, pero añadimos una función para comprobar todos los campos en los comandos get para asegurar que se esta serializando y deserializando correctamente y no se pierden datos.

- Guardar un valor y recuperarlo para comprobar que coincide.
- Intentar obtener un valor de una clave que no ha sido almacenada.
- Intentar almacenar datos con un formato inválido para verificar que se detectan errores.
- Comprobar que no se puede sobrescribir una clave ya establecida.
- Verificar que una clave existe antes y después de eliminarla.
- Almacenar un valor, modificarlo y recuperarlo para confirmar los cambios.
- Intentar modificar una clave que no existe para comprobar que se genera un error.