

TP4 Servidor de IDs Autoincrementales

Autor: Maximiliano Curia
Email: maxy@gnuserservers.com.ar
Padrón: 78719

Resumen

Se implementó un servidor de IDs autoincrementales utilizando conexiones TCP para la comunicación, hilos para manejar múltiples clientes, y mutex para asegurar la consistencia de los datos. Se utilizaron, además, las estructuras de datos *map* y *vector* provistas por la STL de C++, para la manipulación y el almacenamiento en memoria de los datos. Para realizar consultas a los descriptores de archivo de forma no bloqueante se utilizaron las llamadas al sistema *poll*.

Por otro lado, a modo de prueba, se realizó un programa cliente capaz de comunicarse con el servidor de IDs autoincrementales.

Servidor de IDs Autoincrementales

El servidor de IDs, es un pequeño servidor TCP que ante una consulta de una clave válida le devuelve al cliente el siguiente ID (identificador numérico incremental) correspondiente a esa clave.

Al iniciarse, el servidor toma los valores de los IDs actuales para las claves en uso a partir de un archivo. Y al finalizar, escribe el nuevo estado de las claves en el mismo archivo.

El servidor soporta múltiples clientes simultáneos, soporta recibir múltiples pedidos simultáneos del mismo cliente, los cuales serán procesados en el orden de llegada, y además puede ser detenido casi inmediatamente, gracias a las lecturas no bloqueantes que realiza en cada hilo.

La invocación del servidor es de la forma:

```
./server puerto_a_escuchar nombre_de_archivo
```

Cliente de prueba

El cliente es bastante más sencillo, una vez iniciada la conexión con el servidor, espera en su entrada estándar una clave a consultar, esta se la consulta al servidor, espera la respuesta, imprime el ID obtenido y vuelve a esperar la entrada de la siguiente clave.

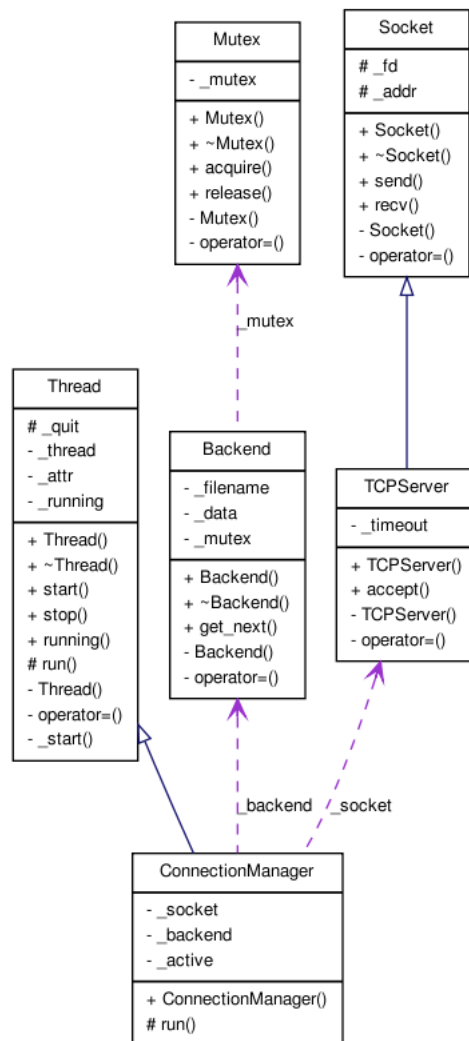
La invocación del cliente es de la forma:

```
./client host_del_servidor:puerto_del_servidor
```

Diseño de la solución

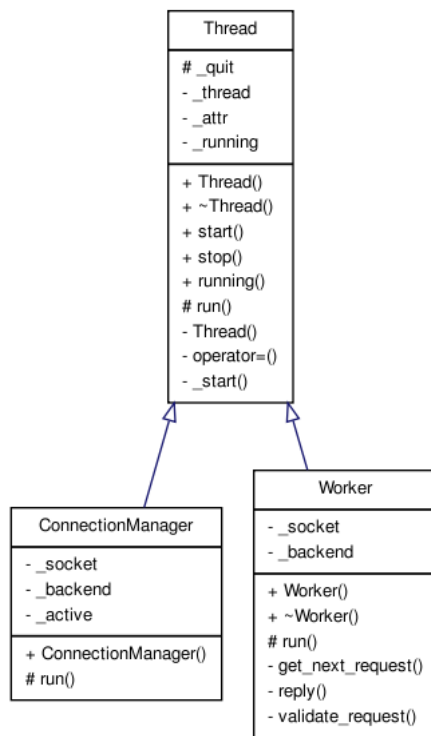
Para diseñar el servidor y el cliente, primero se consideraron las distintas partes del programa y se le dio a cada una de ellas una entidad independiente. Así, se planteó tener clases distintas para **Socket**, **Thread**, **Mutex**, **Backend**.

Las primeras tres son clases muy simples, que básicamente abstraen el uso de las bibliotecas de C para estos fines. Por eso se decidió implementarlas lo más minimalistas posibles, agregándole complejidad sólo cuando el comportamiento esperado no se ajuste al provisto por la capa inferior.



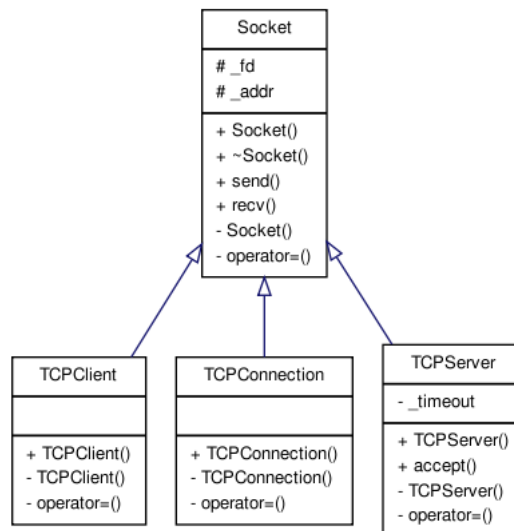
Por otro lado, **Backend**, también resultó ser una implementación minimalista, alrededor de *map*, utilizando un **Mutex** al obtener el siguiente ID.

Para poder atender las comunicaciones con los distintos clientes, se decidió ejecutar en los hilos separados la acción de *accept* y cada una de las conexiones con los clientes. Para ello se crearon dos clases herederas de **Thread**.



La clase **ConnectionManager** es la encargada de recibir las conexiones y cada vez que se acepta una nueva conexión crea un nuevo hilo de tipo **Worker**, que será el encargado de atender las consultas del cliente.

Con respecto al manejo de conexiones de red, se prefirió crear una clase heredera para cada uso de **Socket**, siendo *accept*, por ejemplo, un método sólo existente en **TCPServer**.



De esta forma, cada clase es bastante clara en su uso, un cliente recibe los datos del servidor al cual conectarse en su constructor, un servidor se crea con la información del puerto en que tiene que escuchar y el método *accept* del servidor devuelve una conexión establecida con un cliente.

Al hacer esto, la clase `Socket` quedó bastante reducida, al punto que no tiene sentido una instancia de esta. Sin embargo, podría ser útil tener una colección de `Sockets` que represente las conexiones actuales, ya sean conexiones en modo cliente o en modo servidor, y dado que todas estas comparten la interfaz para el envío y recepción de datos sería transparente para esta colección. Es decir, en esta jerarquía de clases se priorizó una interfaz práctica en el uso, que oculte el comportamiento.

Uno de los problemas más difíciles de solucionar que planteó este trabajo fue la necesidad de poder terminar el servidor correctamente en cualquier momento. Para esto se tuvieron que reemplazar las llamadas bloqueantes por llamadas con un tiempo de expiración, de forma tal que un hilo pueda revisar si debe finalizarse. A su vez, esto implica que una lectura puede no haberse hecho o estar incompleta al momento de procesarla por lo que esto debe señalizarse de alguna manera. El diseño de esa parte del código puede plantearse como:

```
while not quit:
    status, request = get_next()
    if status == COMPLETE:
        process(request)
    elif status == ERROR:
        break
    elif status == INCOMPLETE:
        pass
```

Haciendo que el tiempo máximo de bloqueo sea lo suficientemente bajo como para que al salir del servidor no sea notorio (con una cantidad razonable de hilos), y que tampoco sea significativo para la carga del procesador el procesamiento de este ciclo no bloqueante para una cantidad razonable de hilos, se consiguió el comportamiento deseado.

Conclusiones

Al hacer el trabajo adicional de separar en clases por entidades lógicas estas clases pueden ser usadas como base para otros servidores, procesos con múltiples hilos, etc. Ocultando cierto nivel de complejidad que el uso de estos implica.