75.42 Taller de Programación I

Informe: Ejercicio N° 4

"El código Draka (desencripción por fuerza bruta)"

1. Introducción

Nuestros agentes finalmente han recuperado el código utilizado por los *Draka* para encriptar y desencriptar sus mensajes. Si bien el código es relativamente simple, hasta el momento no ha sido posible encontrarle vulnerabilidades. Solo se sabe que los textos encriptados son ASCII [1] (códigos de carácter menores a 128) y que las claves solo utilizan dígitos ASCII. Aparentemente, debido a esto, la única opción para recuperar el texto será realizar un ataque de *fuerza bruta* [2].

Las claves utilizadas son de longitud suficiente como para hacer que un ataque por fuerza bruta con una sola máquina lleve una excesiva cantidad de tiempo. Por lo tanto, se optará por una solución distribuida siguiendo un esquema cliente-servidor: el servidor se encargará de fraccionar el trabajo y los clientes de resolver cada una de estas partes, enviando los resultados al servidor.

Detalles mas precisos de la problemática y de las condiciones preestablecidas se pueden encontrar en el enunciado del ejercicio[1].

2. Consideraciones de diseño

Para la correcta implementación de la solución fue necesario plantear y establecer cómo se debería comportar el sistema ante ciertas situaciones que no fueron especificadas en el enunciado del problema. A continuación se listan las contemplaciones instauradas:

- Al producirse una solicitud de trabajo por parte de un cliente, si el servidor le indica a este último que no hay trabajo para asignarle, entonces el cliente cerrará la conexión que los vinculaba y dará por finalizada su ejecución;
- El servidor almacenará todas las posibles claves enviadas por los clientes, de manera de que, una vez se indique la detención del mismo, se pueda determinar si existe una ambigüedad o si es única la clave;
- En el envío y en la recepción de datos se considera como fin de mensaje al caractér "$\backslash n$" establecido por el protocolo.
- El programa del lado servidor no finalizará su ejecución a menos que se ingrese "$q\backslash n$" por la entrada estándar.

3. Diseño

Como se mencionó anteriormente, se optará por una solución distribuida siguiendo un esquema cliente-servidor. De esta manera, poseeremos dos implementaciones independientes una de la otra, pero que trabajarán en conjunto una vez establecida la conexión entre las mismas. Por lo tanto, vamos a tener un programa *servidor* y otro programa *cliente*.

En los apartados que siguen pondremos la atención en aquellos aspectos de la implementación de ambos programas que pueden ser relevantes a causa de su complejidad o particularidad. En estos se describen los inconvenientes que presentan y la forma en que fueron resueltos.

[1] Se ha evitado hacer un relevamiento de la totalidad de la información que nos fue conferida, de manera de poder mantener el foco del informe en la forma en que se ha encarado la solución del problema.

3.1. Cliente

Sin duda alguna, este ente del esquema es el menos complejo ya que su tarea se reduce a establecer inicialmente una conexión con el servidor, para luego solicitarle una parte del trabajo a realizar.

En el caso de que el servidor le indique que no hay trabajo disponible, el cliente simplemente finalizará su ejecución. En la situación opuesta, recibirá un mensaje encriptado (codificado por el servidor en hexadecimal para su envío a través del socket) y el rango de claves, siendo su tarea probar cada una de ellas por fuerza bruta, notificando a través del socket cuales son posibles claves.

Finalizadas las pruebas terminará su ejecución automáticamente dando fin a la conexión con el servidor.

Debe tenerse en cuenta que ante cualquier problema surgido en la conexión, ya sea por una interrupción en la misma o por la llegada de mensajes erróneos que no se ajustan al protocolo u orden esperado, el programa cliente también finalizará su ejecución.

Profundizando medianamente, este ente es implementado por la clase *Cliente*, la cual, por las razones expuestas al inicio del apartado, se consideró que no sea un hilo independiente del programa invocante.

3.1.1. Pruebas sobre el *Código Draka*

Desde un principio se ha establecido que el cliente es el encargado de probar el rango de claves, asignado por el servidor, en el código utilizado por los Draka para encriptar y desencriptar sus mensajes.

Para lograr una solución más elegante y mas limpia, se tomó la decisión de encapsular dicho código en una clase de métodos estáticos, *CodigoDraga*, la que a su vez proporciona un método que recibe una clave y que se encarga de realizar la prueba sobre el código Draka, devolviendo como resultado de esta acción un valor lógico, es decir, si retorna "true" significará que la clave pasó la prueba y efectivamente puede ser considerada una posible clave, o si devuelve "false" la clave probada deberá ser descartada.

3.2. Servidor

Para el funcionamiento del servidor se ha propuesto utilizar una lógica muy simple. Esta consiste en tener a la clase *Servidor*, la cual es un hilo de ejecución independiente del hilo principal del main, esperando por solicitudes de conexión por parte de los clientes. Cuando se recibe una petición de conexión por parte de un cliente, el mismo servidor se encarga de crear un objeto de la clase *ConexionCliente*. A este objeto se le pasa el file descriptor correspondiente al nuevo socket que mantendrá en contacto al cliente con este último objeto. Además, el objeto que representa a la conexión será almacenado en una lista de conexiones existentes, para permitir al servidor saber que conexiones debe cerrar cuando llegue el momento de concluir la ejecución.

Como se puede notar, esta clase, considerada la principal del lado servidor, simplemente se limita a la escucha a través de un socket, el cual solo es utilizado con este único fin. Todas las demás tareas de comunicación son derivadas y delegadas a las demás entidades, de las que haremos mención en los siguientes apartados.

3.2.1. Conexión con el cliente

Anteriormente se adelantó que la clase *Servidor*, ante una nueva petición de conexión por parte de un host, creaba un objeto del tipo *ConexionCliente*. Esto significa que este objeto es de ahora en más el único responsable del enlace con el cliente que realizó la petición, es decir, es el único encargado de mantener la comunicación con su par del otro lado del socket. Entre estos se hará el envío de mensajes protocolares los cuales permitirán la asignación de tareas, notificación de posibles claves, etc. Para esto, el ente que administra la conexión se mantendrá en estrecha relación con el *controlador de tareas* del servidor.

3.2.2. Control de tareas

Otro de los entes que se encuentra a las ordenes del servidor, ya que es creado por este mismo y no puede existir mas allá del alcance de este, es el representado por la clase *ControladorDeTareas*. Como su nombre lo indica, es el encargado de administrar las tareas. Más especificamente, su labor está centrada en:

- *División de tareas*: deberá fraccionar las partes de trabajo que deben ser asignadas a cada cliente, como así también indicar cuando ya no hay mas tareas por repartir;

- *Recepción de claves*: cada objeto del tipo *ConexionCliente*, al recibir una clave de parte del cliente notificará al controlador de tareas despachándole esta misma clave;
- *Finalización de tarea*: cuando un cliente envía el mensaje de que ha concluido con su parte del trabajo, el objeto *ConexionCliente* asociado a dicha conexión notificará al controlador de tareas de tal evento;
- *Notificar estado de tareas*: deberá informarle al servidor del estado de las tareas cuando este lo solicite.

De estas cuatro labores debemos destacar una en particular: la *recepción de claves*. Esto se debe a que presenta una de las problemáticas mas interesantes.

Tal como se mencionó, el controlador de tareas recibirá las claves, pero hay que tener en cuenta que cada objeto *ConexionCliente* se encuentra corriendo en un hilo independiente y que se puede dar el caso de que dos de estos hilos traten de notificar al mismo tiempo la llegada de una nueva clave. Es aquí donde se decidió convertir a la clase *ControladorDeTareas* en thread-safe, es decir, con soporte para multithreading. Por lo tanto, se incluyó una auto protección (un objeto de tipo *Mutex*) en la clase, lo cual evita que las instancias de esta misma sean accedidas al mismo tiempo por hilos diferentes.

Por otro lado, al ser informado de la llegada de una nueva clave, el controlador de tareas debe tomar a esta última e ingresarla en una lista proporcionada por la clase *Servidor*. Esta lista presenta problemas semejantes con respecto al multithreading, razón por la cual será discutida en el apartado que sigue, dándose detalles de cómo es que se evitan los conflictos de acceso a un objeto por parte de múltiples hilos.

3.2.3. Clase *Lista*

Como ya es sabido, la *STL* [3] no proporciona estructuras con soporte para multithreading. Este es el caso del contenedor *list*, el cual no es *thread-safe*. Ante la necesidad de la utilización de esta en el lado del servidor, de manera de poder albergar a las claves que arriven desde los clientes (como se explicó en el apartado anterior), es que se decidió extender a *list* y agregarle el debido soporte.

Para esto, hemos creado una clase *Lista*, la cual utiliza internamente al contenedor *list*, pero con la salvedad de que cada método se encuentra protegido de los múltiples accesos posibles desde distintos threads.

Es así que se estableció como un atributo interno y privado un objeto de tipo *Mutex*, el cual es bloqueado en cada invocación a los métodos, evitando que mientras se ejecutan las instrucciónes de este último, otros threads puedan hacer uso de la lista, quedando en su defecto a la espera de la liberación del mutex.

3.3. Protocolo y comunicación

Respecto a la forma en que se comunican los clientes y el servidor, ya se adelantó anteriormente la utilización de un protocolo fijo e irremplazable. Para poder unificarlo y que los objetos comprendan los mismos tipos de mensajes se creó la clase *Protocolo*, la cual simplemente alberga constantes que definen los distintos tipos de avisos a intercambiar entre los entes.

Por otro lado, para abstraer la forma de enviar y recibir datos, se decidió establecer como intermediario a la clase *Comunicador*, la cual se encarga de emitir o de recibir datos (a través del socket proporcionado por su empleador) teniendo en cuenta el protocolo antes mencionado.

4. Futuras mejoras

Se listan aquí posibles mejoras a realizar en el futuro, como así también, ciertas falencias que fueron descubiertas habiéndose hecho ya la entrega del producto, a fin de denotar conciencia sobre el trabajo realizado:

- Mejorar informe de errores en clase *Cliente*, de manera de proporcionar información mas detallada acerca de la desconexión del socket;
- Clases *Thread* y *Mutex* mas completas con respecto al soporte de mas opciones;
- Mejorar empaquetamiento de la clase *Comunicador* y el armado de mensajes por parte de este.

5. Esquema del diseño

A continuación, en la *Figura 1*, se ilustra el diagrama de clases principal, donde se pueden ver las entidades que intervienen tanto en el servidor como así también en el cliente. Cabe destacar que se han omitido, para mayor comprensión, ciertas clases secundarias que hacen a la totalidad del sistema, pero que su presencia no es indispensables para el entendimiento del funcionamiento general del mismo.

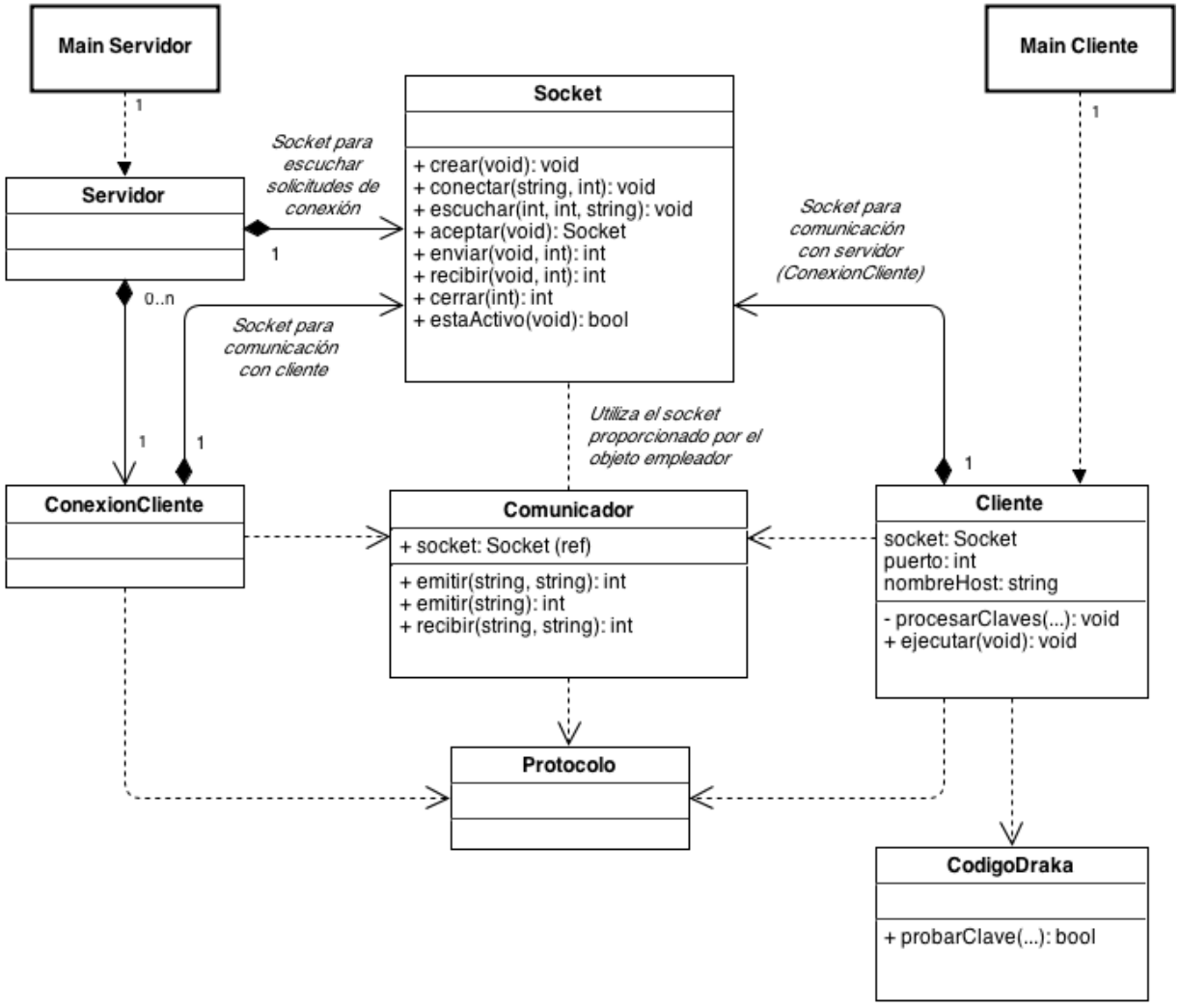

Figura 1: *Diagrama de clases principal del cliente-servidor.*

Por último, en la *Figura 2*, se muestra un diagrama de clases ampliado del lado servidor. En este se ilustra qué clases son un *Thread*, es decir, que clases proveen objetos que corren en distintos hilos al ser instanciadas. Además se puede observar la relación entre entidades, las cuales fueron descriptas en secciones anteriores.

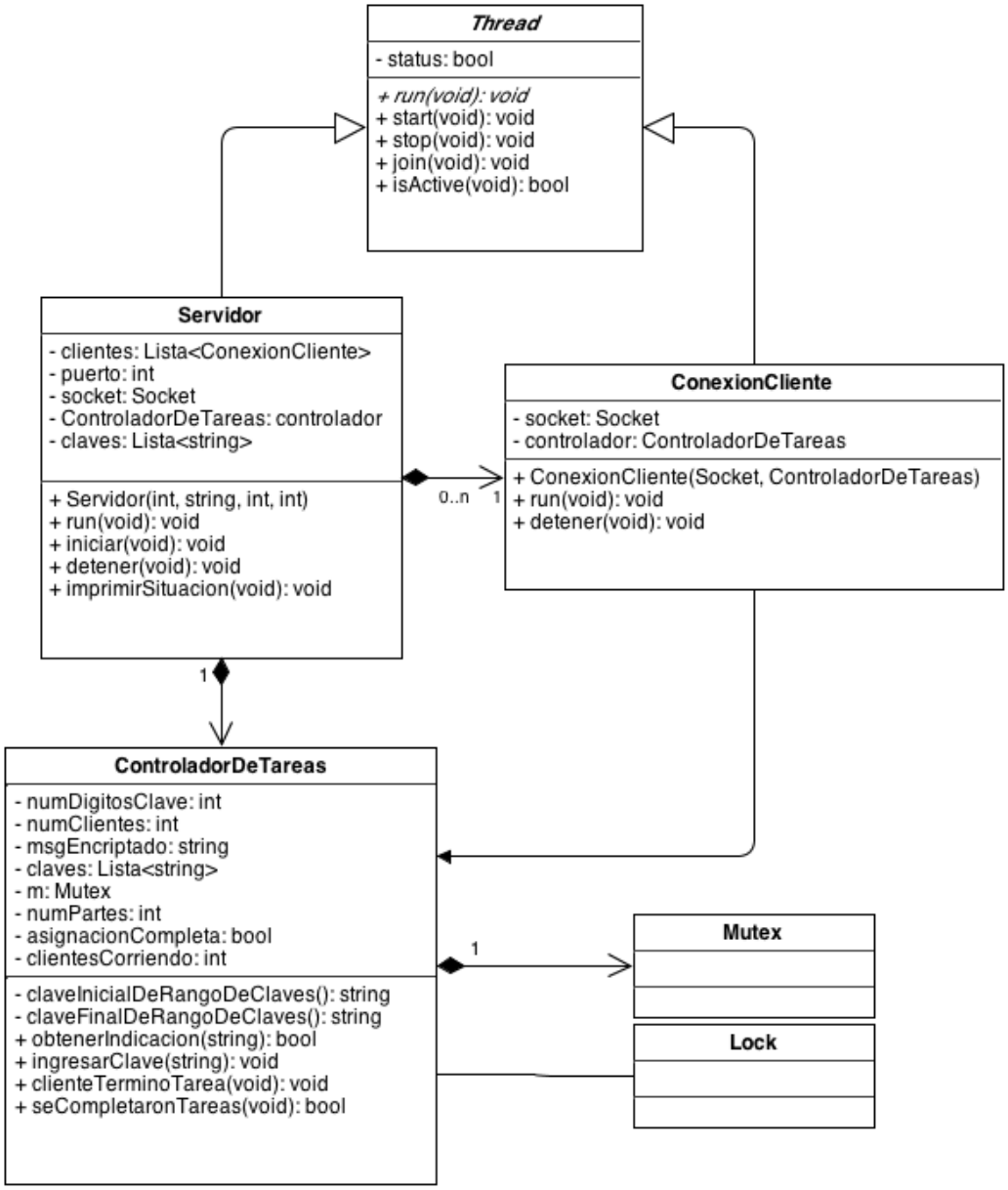

Figura 2: *Diagrama de clases del servidor.*

Referencias

[1] Código ASCII, `http://en.wikipedia.org/wiki/ASCII`

[2] Ataque por Fuerza Bruta (Brute-force attack), `http://en.wikipedia.orig/wiki/Brute-force_attack`

[3] Standard Template Library, `http://en.wikipedia.org/wiki/Standard_Template_Library`