Reto 1 - Chat

Por:
Andrés Mejía
200710012010

Santiago Achury
200810050010

Profesor:
Edwin Montoya

Universidad EAFIT

2011-2

Indice

Análisis y diseño

Arquitectura

La arquitectura que utilizaremos es Cliente/Servidor. Nuestra aplicación implementa su propio protocolo de chat. Decidimos implementar esta opción (y no MOM ó invocación remota) porque nos pareció la manera más simple e intuitiva de estructurar la aplicación.

Decidimos utilizar el protocolo TCP. Nos pareció más apropiado que UDP porque podemos basarnos en algunas de las características de TCP para simplificar nuestro protocolo de chat. Más concretamente:

- TCP garantiza que todo paquete es recibido por el destinatario. Esto hace más fácil el proceso de enviar un mensaje a la sala de chat, porque no es necesario esperar una confirmación para estar seguros de que el servidor recibe el mensaje.
- TCP tiene estado. De esta manera podemos asociar un usuario en la sala de chat a un socket TCP, y podemos estar seguros de que el usuario que utiliza ese socket siempre es el mismo.

Escogimos el lenguaje de programación Ruby por su simpleza y elegancia.

Evaluemos ahora algunos aspectos sobre nuestro diseño:

Interacción sincrónica/asincrónica

Nuestra aplicación utiliza interacción asincrónica: un cliente envía un mensaje al servidor y el servidor le responde, pero no se garantiza que el servidor responda instantáneamente. El cliente no debe bloquearse esperando la respuesta del servidor, sino que debe seguir su ejecución normal.

La manera como se maneja esto en el cliente es que existen dos hilos: el primero se encarga de leer el texto que escribe el usuario por el teclado y de enviarlo al servidor; el segundo se encarga de leer respuestas del servidor. Como son dos hilos independientes, el cliente no se bloquea cuando no recibe una respuesta instantánea del servidor.

Interacción simétrica/asimétrica

Nuestra aplicación utiliza interacción simétrica: tanto el cliente como el servidor puede iniciar la transmisión de un mensaje.

Por supuesto, al tratarse de arquitectura Cliente/Servidor, sólo el cliente puede **iniciar** una conexión, pero una vez se haya hecho esto la conexión permanece abierta y el servidor puede tomar la iniciativa de enviar un mensaje al cliente (de hecho, es así como se transmiten mensajes de chat que han escrito otros usuarios; el servidor le avisa a cada cliente que hay un mensaje nuevo). En este caso nos pareció más conveniente utilizar interacción simétrica que asimétrica porque si fuera asimétrica el cliente tendría que estar haciendo *polling* permanente del servidor. Esto nos parece un inconveniente innecesario y visto que tenemos las dos opciones, decidimos eliminarlo.

Manejo o no de sesión y estado

Nuestra aplicación maneja sesión y estado aunque de una manera muy simple: Cuando se abre una conexión al servidor, el servidor pide el nickname del usuario que acaba de conectarse. En ese momento, el servidor asocia ese nickname con ese socket en una tabla de hash y le informa a todos los demás clientes conectados que alguien acaba de entrar a la sala. En adelante, cuando se recibe un mensaje público, el servidor lo retransmite a todos los sockets presentes en la tabla de hash; y cuando se recibe un mensaje privado el servidor busca en la tabla de hash el socket asociado con el nickname del destinatario del mensaje privado y retransmite el mensaje únicamente por ese socket.

Cuando el cliente cierra el socket, el servidor elimina la entrada correspondiente en la tabla de hash y le informa a los demás clientes el nickname de quien acaba de desconectarse.

Este mecanismo funciona porque al usarse TCP como protocolo de transporte se puede dejar la conexión abierta entre cada cliente y el servidor. TCP nos garantiza que todo lo que escribamos por algún socket será recibido siempre por la misma entidad en el otro lado. De esta manera podemos despreocuparnos de implementar un mecanismo más complejo de sesión.

Modelo de manejo de fallos

El manejo de fallos en nuestra aplicación es relativamente simple: cuando el servidor recibe un mensaje que desconoce (es decir, que no está presente en nuestro protocolo), responde siempre con un mensaje que indica que se envió un comando desconocido. Más precisamente, responde con:

```
666 Invalid command. Ignoring.
```

El cliente, en cambio, si recibe un mensaje desconocido, simplemente lo ignora por completo.

Tanto en el cliente como en el servidor se hace un manejo apropiado de excepciones para evitar que alguno de los termine inesperadamente a causa de una excepción.

Modelo de seguridad

Nuestra aplicación realmente no implementa un modelo estricto de seguridad. La única revisión que se hace es que un cliente no intente identificarse como otro que ya está en la sala para suplantarlo.

Es cierto que los mensajes privados podrían llegar a interceptarse por una máquina que esté en la misma red que la máquina del destinatario puesto que se trata de paquetes TCP (y existen técnicas conocidas para interceptar paquetes TCP dentro del mismo segmento de red). Es posible resolver este problema implementando un esquema de encriptación al estilo SSL, por ejemplo, pero esto complica las cosas de manera considerable y pensamos que se sale del alcance de esta práctica.

Niveles de transparencia

Al usarse un protocolo de chat, se agrega un nivel de transparencia pues se puede cambiar el cliente por cualquier otra aplicación que implemente el protocolo. Para los usuarios conectados es indiferente qué cliente están usando los otros usuarios. Si en el futuro se implementa un cliente con una interfaz gráfica, podrán comunicarse ambos clientes transparentemente a través del servidor.

Multiusuario

Para ofrecer soporte multiusuario, nuestro servidor crea un hilo por cada conexión que recibe. Este hilo se mantiene abierto mientras la conexión TCP subyacente permanezca abierta.

Atributos de calidad

Nuestra aplicación debe satisfacer los siguientes criterios de calidad:

- Cuando un mensaje de chat se envía, debe llegar intacto a los destinatarios. El servidor no debe modificarlo en absoluto.
- Los mensajes privados no deben ser retransmitidos a una persona que no sea su destinatario.
- Los mensajes públicos deben ser retransmitidos a todos los usuarios que estén conectados al servidor en ese momento.
- El servidor debe soportar varios clientes simultáneamente puesto que no tiene sentido iniciar una conversación de chat de un solo usuario.
- El proceso del servidor no debe terminar aunque haya una excepción, para evitar que se caiga el servicio de chat.

Consideraciones de escalabilidad y extensibilidad

No podemos afirmar que la manera como está implementado el servidor es muy escalable. El servidor lanza un nuevo hilo cada vez que recibe una conexión y este esquema tarde o temprano alcanzará los límites físicos de la máquina donde corre el servidor. Si quisiéramos implementar un servidor más escalable tendríamos que recurrir a otros métodos como por ejemplo *pool* de conexiones ó un mecanismo que permita que varios servidores compartan la misma sala de chat (esto permitiría tener varias máquinas diferentes compartiendo la misma sala de chat detrás de un balanceador de carga).

En términos de extensibilidad, si quisiéramos agregar una nueva funcionalidad (por ejemplo varias salas de chat al mismo tiempo) las instrucciones a seguir son modificar el protocolo y después implementar los cambios en el cliente y el servidor.

Protocolo Chatte

Ingreso a la sala

Se abre una conexión TCP al servidor.

El servidor responde con:

```
0 Hello, nickname please.
```

El 0 es una petición al cliente para que envíe su apodo. El cliente responde con:

```
Achury
```

El servidor responde con una de tres opciones:

1. El nickname es válido

```
200 Welcome to the chatte.
```

En este caso la conexión permanece abierta y se puede proceder a enviar y recibir mensajes.

2. El nickname es inválido

```
201 Invalid nickname. Please try again. Bye.
```

En este caso el servidor cierra la conexión. Los nicknames válidos satisfacen la expresión regular `/\A[a-z0-9\._+-]+\z/i` (nótese que el nickname no puede contener espacios).

3. El nickname ya está siendo utilizado

```
202 Nickname already in use. Please try with another one. Bye.
```

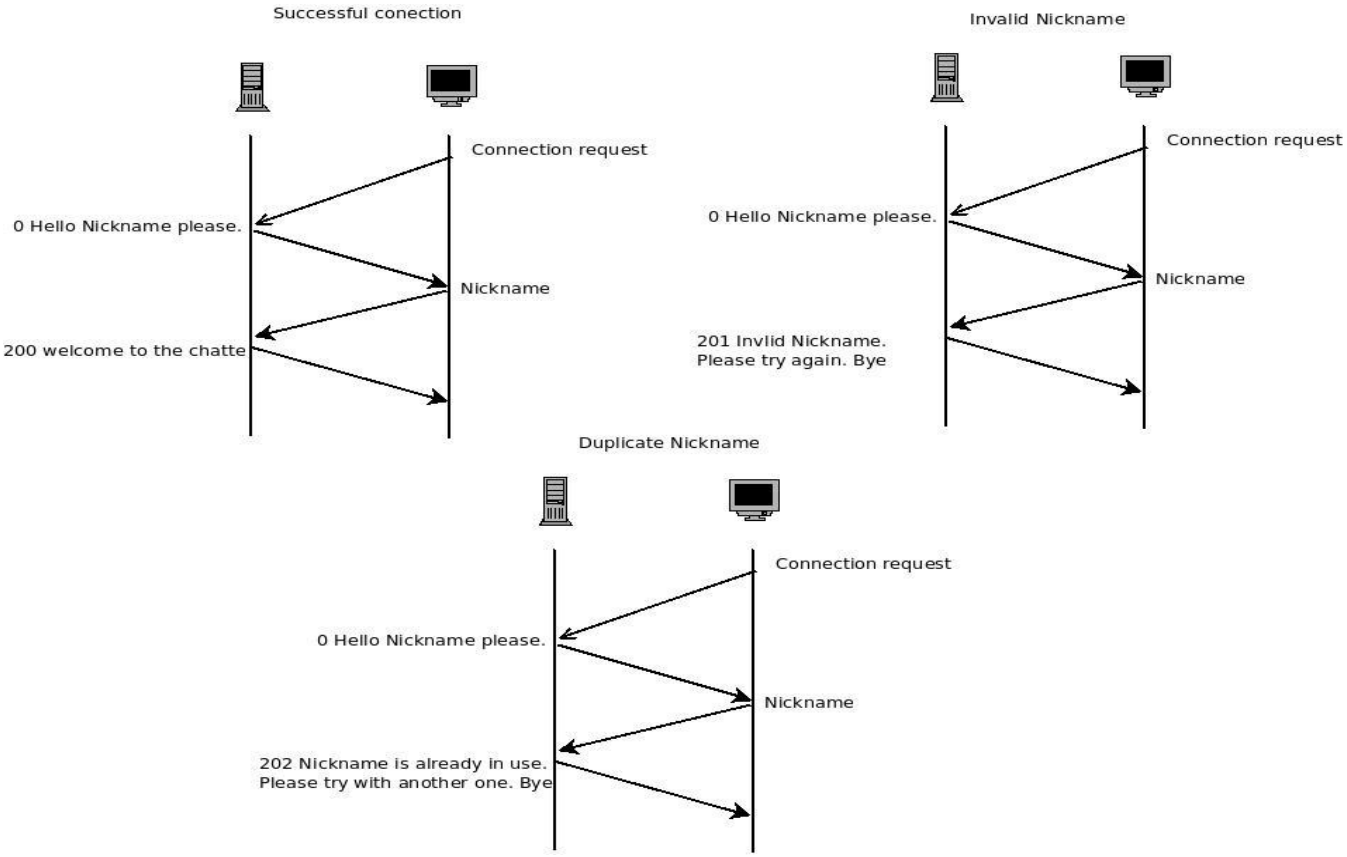
En este caso el servidor cierra la conexión.

Cuando un usuario entra al chat el servidor notifica a los demas usuarios activos

```
150 nickname just joined to the chat
```

cuando un usuario sale del chat el servidor notifica a los demas usuarios activos

```
151 nickname just left the chat
```



Envío de mensajes

Para enviar un mensaje a toda la sala en general, el cliente debe enviar:

```
100 Hola, acabo de llegar. ¿Hay alguien aquí?
```

Todo lo que venga después de `100` será retransmitido a todos los usuarios conectados.

El servidor responde con

```
101 Message sent.
```

Para enviar un mensaje privado, el cliente debe enviar:

```
102 Andy Andy, ¿ya hiciste la tarea de telemática?
```

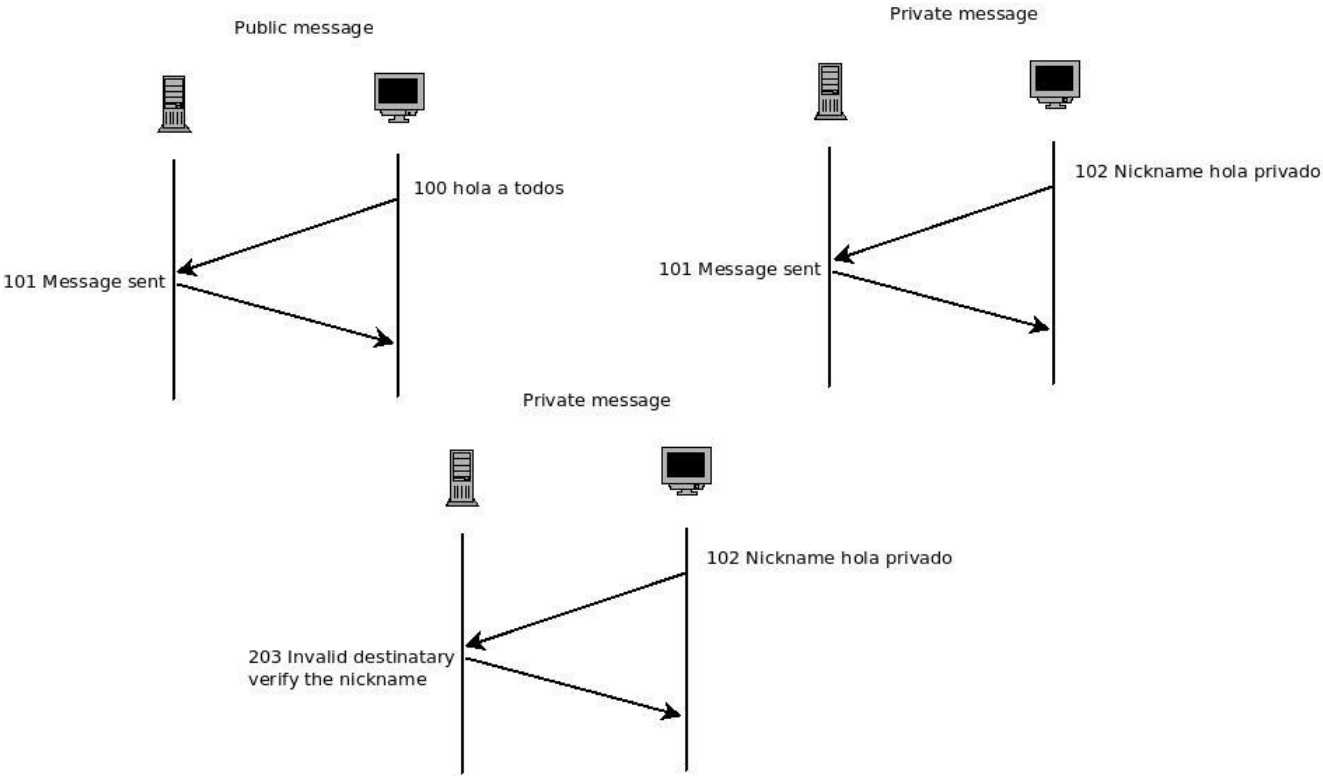
La primera palabra después del código es el nickname del usuario destinatario del mensaje privado. De ahí en adelante es el mensaje a enviar.

Si el usuario destinatario no está conectado, el servidor responde con

```
203 Invalid destinatary. Verify the nickname.
```

Si el mensaje privado fue enviado, el servidor responde con

```
101 Message sent.
```



Recepción de mensajes

Para que el cliente reciba un mensaje público, el servidor le envía:

```
100 Andy Hola, ¿hay alguien aquí?
```

El primer token después del código es el nickname de la persona que envió el mensaje. En adelante es el mensaje.

El cliente debe responder con

```
101 Message received.
```

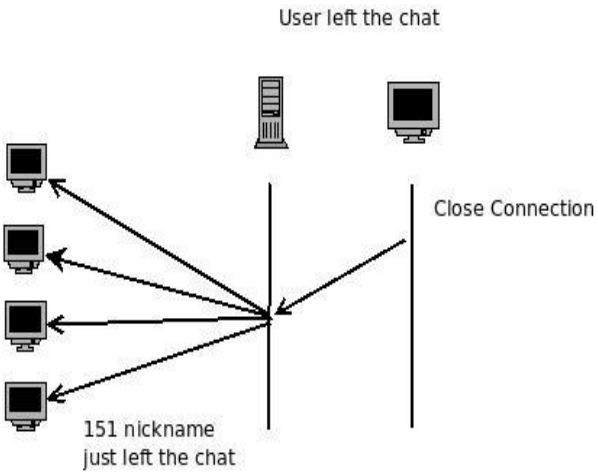
Para que el cliente reciba un mensaje privado, el servidor le envía:

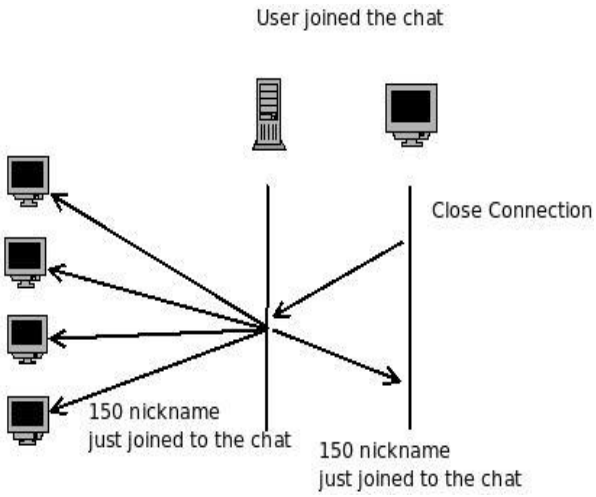
```
102 Andy Achury, sí, ya hice la tarea.
```

El primer token después del código es el nickname de la persona que envió el mensaje privado. En adelante es el mensaje.

El cliente debe responder con

```
101 Message received.
```





Instalación y ejecución

Instalación de Ruby

La aplicación está escrita en el lenguaje de programación Ruby. Esto implica que se necesita instalar la máquina virtual de Ruby para poderla ejecutar.

La manera más fácil de instalar Ruby es usando RVM (Ruby Version Manager). Instrucciones detalladas sobre como instalar RVM se encuentran en http://beginrescueend.com/rvm/install/.

Una vez instalado RVM, debe instalarse la versión más reciente de Ruby 1.9.2. Esto puede hacerse ejecutando el comando:

```
$ rvm install ruby-1.9.2-head
```

Después puede comprobarse que se instaló correctamente Ruby 1.9.2, corriendo el comando:

```
$ ruby -v
```

La salida debería ser algo como esto (puede variar según el sistema operativo):

```
ruby 1.9.2p180 (2011-02-18 revision 30909) [x86_64-
darwin10.7.0]
```

¡Listo! Ya se puede ejecutar el servidor.

Ejecución del servidor

El servidor recibe un parámetro desde la línea de comando correspondiente al puerto dónde debe escuchar. Se ejecuta corriendo el comando:

```
$ ruby server.rb 1234
Starting Chatte server. Listening for connections on port
1234...
```

En este momento el servidor se encuentra aceptando conexiones entrantes y se puede continuar a ejecutar el cliente.

Ejecución del cliente

El cliente recibe dos parámetros desde la línea de comandos: el host y el puerto del servidor. Luego el cliente lee el nickname del usuario desde la entrada estándar y se conecta a dicho host y puerto. Veamos un ejemplo:

```
$ ruby client.rb localhost 1234
Enter your nickname:
Andy
Connecting...
Andy just joined the chatte
```

Terminación del servidor

Para terminar el servidor, hay que enviarle la señal SIGINT. Esto puede hacerse presionando las teclas Ctrl+C en la consola donde está corriendo el servidor (en sistemas Unix).

Terminación del cliente

Para terminar el cliente, se le puede enviar la señal SIGINT similiar a como se hace con el servidor o se puede escribir `/quit` ó `/exit` en el cliente.