Práctica 6

Sockets Servidores

Programación Orientada a Objetos

Humberto Alejandro Ortega Alcocer

Opción: 3 “Tablero de avisos electrónico”

13 de Enero de 2021

Grupo: 2CM1

Introducción

Para entender completamente el paradigma cliente-servidor, debemos comenzar por desarrollar alguna aplicación que muestre los beneficios de dicho paradigma. Es importante entender que el objetivo es permitir a múltiples clientes interactuar mediante un tercero (el servidor) que, de forma centralizada, concentrará la información que alimenta el servicio.

Un tablero de mensajes es un ejemplo práctico de este esquema. Además de ser un programa simple y fácil de entender, es, en esencia, uno de los más antiguos usos que se le ha dado a la comunicación en red entre computadoras. Para utilizarlo, basta con solicitar la lista de mensajes existentes en el servidor, filtrar alguno en particular, o bien, añadir un nuevo mensaje. La idea general es que los mensajes que ahí se muestran pueden ser añadidos por cualquier usuario, y cualquier usuario, a su vez, puede consultar los mensajes. Los mensajes están compuestos por un título y un cuerpo, lo cual nos permite ofrecer la funcionalidad de filtrar mediante el título para realizar una búsqueda particular.

Los tableros de mensajes, como se mencionó anteriormente, no son algo nuevo y, en sus inicios, servían para publicar información dentro de universidades para mantener actualizados a los alumnos y profesores sobre noticias e información de interés. Con el tiempo, estos tableros de mensajes evolucionaron en lo que hoy conocemos como redes sociales, inclusive uno puede ver algo tan moderno como Twitter y encontrar algunas semejanzas con los simples tableros de mensajes. Otra evolución notable de los tableros de mensajes son los foros, dónde se extiende su funcionalidad para permitir que se creen mensajes y discusiones subsecuentes sobre cada uno.

Si bien, el objetivo de la práctica consiste en realizar un tablero de mensajes que permita a varios clientes consultar los mensajes y añadir mensajes, es fácil ver

hasta dónde se puede llevar el concepto y la relevancia de éste en cuanto se empieza a vislumbrar la oportunidad de desarrollar algo aún más complejo.
El planteamiento de la práctica, en palabras del profesor Roberto Tecla, es:

*3.-**Tablero de avisos electrónico**
Codificar un cliente y un servidor para ofrecer un servicio de Tablero de avisos electrónico. Dicho programa tendrá 2 comandos setinfo y getinfo

java TableroAvisos setinfo titulo aviso
java TableroAvisos getinfo titulo o java TableroAvisos getinfo alltags

Un cliente envía un aviso y su titulo al servidor este lo almacena en un HashMap (o en dos arreglos). El cliente puedes solicitar un aviso del servidor por título o pedir los títulos de todos los avisos. Si el cliente envía el titulo de un aviso el servidor lo usa para buscar en el arreglo de avisos el aviso correspondiente a dicho título.

ejemplos

(se lavan alfombras) (a domicilio precios modicos)
(se vende perro) (cachorro dalmata sabe decir te amo)
(le metemos el migajon a sus bolillos) (no sufra mas nosotros lo hacemos por ud.)
(disciplinamos a sus hijos) (academia militar con 50 años de experiencia)

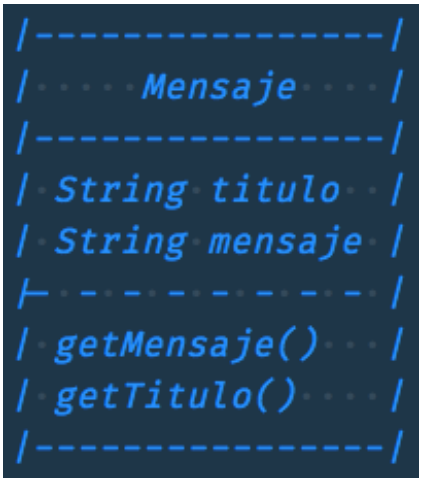
Desarrollo

Para comenzar con el desarrollo de la práctica, debo mencionar que me he apoyado de algunos archivos y funcionalidades presentes en la práctica anterior (Sockets Cliente: Nano Alexa) para lo que refiere a comunicación en red. Algunos de los archivos que he reutilizado son:

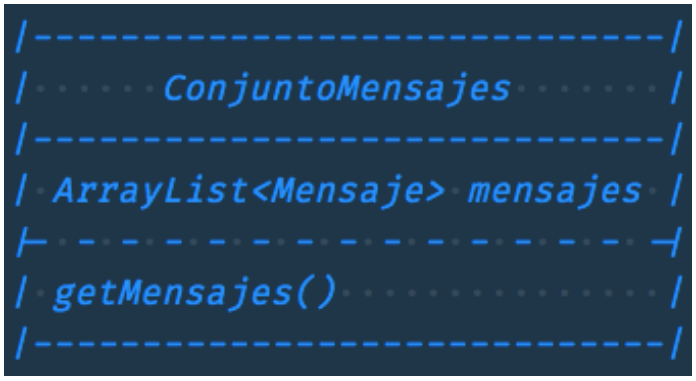
- Red.java
- LeeRed.java
- IncomingReader.java

Estos archivos han sido modificados únicamente con fines estéticos (indentación, comentarios y algunos logs para tareas de debug), por lo que no ahondaré en su implementación y explicaré los archivos que describen el comportamiento del programa.

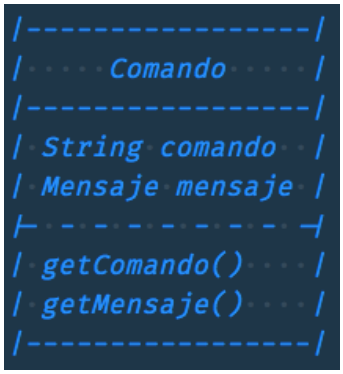
Comenzaremos por visualizar las distintas clases que representan objetos de comunicación entre el cliente y el servidor, así como el objeto que almacenará el mensaje. Primero que nada, la clase Mensaje, que representa un mensaje dentro del tablero, cuyo diagrama UML se muestra a continuación:



Como podemos observar, Mensaje incluye un título y el mensaje en sí. Esta clase implementa la clase Serializable por lo que el contenido del objeto puede ser enviado de forma segura a través de la red mediante el uso de sockets. Una clase derivada de esta es ConjuntoMensajes, la cual únicamente representa, como su nombre lo dice, un conjunto de mensajes, y contiene un ArrayList con los mensajes, siendo su diagrama UML:



Como se puede observar, es una clase muy trivial que únicamente ayuda para la tarea de transmisión de múltiples mensajes de forma adecuada. La última clase que representa objetos de comunicación es Comando. En la clase Comando se incluye una variable para almacenar el comando a realizar, esta clase nos sirve para especificar la tarea que solicita el cliente del servidor, de esta forma el cliente puede elegir entre las 3 distintas opciones que ofrece el servidor. El diagrama UML se muestra a continuación:



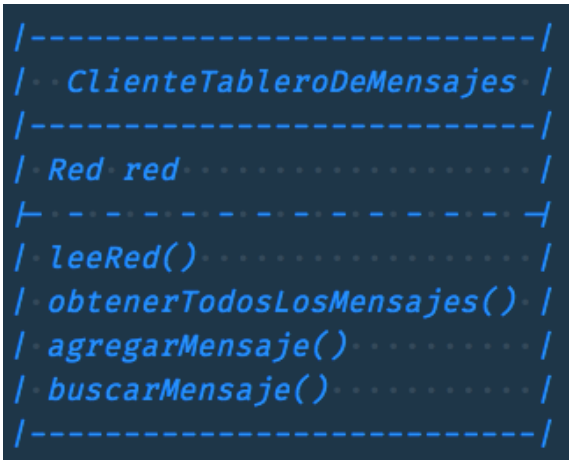
Esta clase, sirve para que el cliente pueda emplear alguna de las 3 rutas disponibles de interacción con el servidor:

- Añadir un nuevo mensaje al tablero
- Buscar un mensaje en el tablero usando el título
- Obtener todos los mensajes presentes en el tablero

Si bien, esta clase podría ser sustituida por variables “crudas” enviadas mediante la red, preferí encapsularlas dentro de la clase por temas de legibilidad de código. Ahora comenzaremos analizando el servidor, para ello, usé de base el archivo VerySimpleChatServer.java, el cual fue proporcionado por el profesor para ser usado en una práctica anterior por lo que mostraré las secciones que fueron modificadas. Comenzaremos por visualizar la sub-clase ClientHandler, en la cual ocurre el proceso de control para cada uno de los clientes conectados al servidor. Cada que recibimos una conexión entrante, aceptamos la conexión y creamos un hilo que escucha por la interacción del cliente con el servidor, de esta manera nuestro servidor puede permanecer aceptando nuevas conexiones sin preocuparse por el manejo y la interacción con cada cliente. En esencia, creamos un flujo de entrada y otro de salida con el socket del cliente que representa la conexión aceptada y, en cuanto recibimos un Comando del cliente, discernimos la operación en función del mismo y la realizamos. En la descripción del procedimiento se puede visualizar más a detalle la implementación de la lógica pero realmente consiste en una serie de condiciones que determinan el camino a seguir. Si se desea agregar un mensaje, éste vendrá incluido dentro del Comando y será añadido inmediatamente. Si se desea realizar una búsqueda, se llamará a una función particular que se encarga de buscar dentro de todos los mensajes presentes en el servidor hasta encontrar un título que sea igual al que el usuario nos ha proporcionado. Finalmente, si lo que se desea es obtener el listado de todos los mensajes, el servidor envía todos los mensajes.

```java
/**
 * Clase para el manejo individual de cada uno de los clientes.
 */
public class ClientHandler implements Runnable {
  ObjectInputStream reader;
  ObjectOutputStream writer;
  Socket sock;

  // Constructor de la clase.
  public ClientHandler(Socket clientSocket) {
    try {
      sock = clientSocket;
      reader = new ObjectInputStream(sock.getInputStream());
      writer = new ObjectOutputStream(sock.getOutputStream());
    } catch (Exception ex) {
      System.out.println("Excepción creando ClientHandler:");
      ex.printStackTrace();
    }
  }

  // Método que escucha y procesa los comandos entrantes.
  public void run() {
    // El comando que recibimos del cliente.
    Comando comando;

    // Ejecutamos el ciclo infinito dentro de un try/catch.
    try {
      // Leemos el comando recibido.
      comando = (Comando) reader.readObject();

      // Verificamos el tipo de comando a procesar.
      switch (comando.getComando()) {
        case "AGREGAR":
          // En caso de que sea el comando "AGREGAR", añadimos el mensaje al tablero.
          mensajesEnElTablero.add(comando.getMensaje());
          System.out.println("Se añadió el mensaje: " + comando.getMensaje().getTitulo() + " ~ "
              + comando.getMensaje().getMensaje());
          break;
        case "BUSCAR":
          // En caso de que sea el comando "BUSCAR", buscamos el mensaje con el título
          // proporcionado y lo enviamos de vuelta.
          Mensaje mensajeABuscar = comando.getMensaje();

          // Realizamos la búsqueda.
          Mensaje mensajeEncontrado = buscarPorTitulo(mensajeABuscar.getTitulo());
          ArrayList<Mensaje> temporal = new ArrayList<Mensaje>();
          temporal.add(mensajeEncontrado);

          // Enviamos el mensaje encontrado (aunque sea un vacío).
          try {
            this.writer.writeObject(new ConjuntoMensajes(temporal));
            this.writer.flush();

            System.out.println("Se ha enviado el mensaje que hizo match con \"" + mensajeABuscar.getTitulo() + "\": "
                + mensajeEncontrado.getTitulo() + " ~ " + mensajeEncontrado.getMensaje());
          } catch (Exception e) {
            System.out.println("Excepción enviando el mensaje encontrado:");
            e.printStackTrace();
          }
          break;
        case "TODOS":
        default:
          // En caso de que sea el comando "TODOS" o sea un comando no reconocido
          // enviamos todos los mensajes en el tablero.
          try {
            this.writer.writeObject(new ConjuntoMensajes(mensajesEnElTablero));
            this.writer.flush();

            System.out.println("Se enviaron todos los mensajes correctamente.");
          } catch (Exception e) {
            System.out.println("Excepción enviando todos los mensajes:");
            e.printStackTrace();
          }
          break;
      }

      // Cerramos el socket.
      sock.close();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Como se puede observar, el comportamiento de la clase representa un modelo muy simple pero funcional para los objetivos de la práctica.
Fuera de ésta clase, el servidor realmente no cuenta con mucha más lógica ni operaciones relevantes a considerar. Todos los mensajes recibidos se añaden al arreglo de Mensajes, y se realizan las operaciones correspondientes según el Comando recibido.
Pasando del lado del cliente, la funcionalidad es muy simple, recibimos parámetros desde la terminal, determinamos si los parámetros son “setInfo” o “getInfo” y a partir de ahí realizamos la operación deseada por el usuario.
El diagrama UML de la clase se muestra a continuación:



Como se puede observar, la clase es muy simple. El objeto Red es un objeto que pertenece a una clase previamente proporcionada por el profesor dónde las únicas modificaciones realizadas han sido fijar la dirección IP del servidor a localhost, y se añade un método para finalizar la ejecución del hilo de escucha para permitir liberar el socket y, finalmente, poder terminar la ejecución del programa.
La función “leeRed” es llamada en cuanto recibimos una respuesta del servidor, por lo que su implementación únicamente recibe un ConjuntoMensajes e imprime uno a uno en la terminal.

La implementación de dicha función es la siguiente:

```java
public void leeRed(Object objeto) {
  // Extraemos el objeto.
  ConjuntoMensajes conjuntoMensajes = (ConjuntoMensajes) objeto;
  ArrayList<Mensaje> mensajes = conjuntoMensajes.getMensajes();

  // Imprimimos los mensajes.
  Iterator<Mensaje> iterador = mensajes.iterator();

  while (iterador.hasNext()) {
    Mensaje mensajeActual = iterador.next();

    if (mensajeActual.getTitulo().length() > 0 && mensajeActual.getMensaje().length() > 0) {
      System.out.println(mensajeActual.getTitulo() + " ~ " + mensajeActual.getMensaje());
    }
  }

  // Cerramos la red.
  this.red.cerrarRed();
}
```

Y la implementación de las funciones con los comandos es la siguiente:

```java
// Método para obtener todos los mensajes en el servidor.
public void obtenerTodosLosMensajes() {
  this.red.escribeRed(new Comando("TODOS", new Mensaje("", "")));
}

// Método para agregar un mensaje al servidor.
public void agregarMensaje(String titulo, String msj) {
  this.red.escribeRed(new Comando("AGREGAR", new Mensaje(titulo, msj)));

  this.red.cerrarRed();
}

// Método para buscar un mensaje en el servidor.
public void buscarMensaje(String titulo) {
  this.red.escribeRed(new Comando("BUSCAR", new Mensaje(titulo, "")));
}
```

Si bien con estas funciones es suficiente para realizar la comunicación deseada entre el cliente y el servidor, en el método principal del programa es dónde se incluye el manejo de los parámetros proporcionados en la terminal al programa y la llamada a las funciones correspondientes:

```
// Método principal.
public static void main(String args[]) {
  // Creamos una instancia del cliente.
  ClienteTableroDeMensajes cliente = new ClienteTableroDeMensajes();

  // Verificamos el comando proporcionado.
  if (args.length > 0) {
    if (args[0].equals("setInfo")) {
      // Obtenemos los datos de los argumentos.
      String titulo = args[1];
      String mensaje = args[2];

      // Enviamos el comando.
      cliente.agregarMensaje(titulo, mensaje);
    } else {
      // El comando es "getInfo"
      if (args[1].equals("alltags")) {
        // Todos los mensajes.
        cliente.obtenerTodosLosMensajes();
      } else {
        // Buscar un mensaje particular.
        cliente.buscarMensaje(args[1]);
      }
    }
  } else {
    cliente.obtenerTodosLosMensajes();
  }

  // Finalizamos la ejecución.
  return;
}
```

Un ejemplo de la correcta ejecución del programa en general es:

Conclusión

Cuando comenzamos a trabajar con aplicaciones que interactúan con otros servicios y computadoras mediante una red es cuando realmente se puede visualizar lo complejo que puede llegar a ser un servicio distribuido, inclusive uno tan simple como un tablero de avisos. Si bien mucho de lo que se desarrolló no conlleva una complejidad mayor, esto es porque se omiten muchas consideraciones que deberíamos tomar en un ambiente de producción para el mundo real. Una aplicación de esta índole deberá contar con todas las validaciones y medidas de seguridad necesarias para robustecer su operación y es claro que para los fines de esta práctica no es necesario considerar dichas complejidades, aún así durante el desarrollo de la aplicación uno puede apreciar los distintos puntos débiles de la misma y pensar para uno mismo: "¿Pero qué pasaría si...?", y es ahí dónde se puede reflexionar, tanto como uno quisiera, en torno a lo mucho que ha avanzado el ecosistema tecnológico para abstraer dichas complejidades y permitirnos hacer más con menos código. Esta práctica es una excelente manera de comenzar a ver cómo podemos estructurar una aplicación que hace uso de una red, así como entender la razón de existir de muchas otras herramientas. Uno de los puntos clave que pude observar es que, dado que los mensajes se almacenan en la memoria del servidor, si se llega a dar una excepción que no sea manejada apropiadamente, se pierde toda la información. La solución puede venir en formas distintas como usar archivos, separar en múltiples hilos de ejecución con esquemas de validación y protección de los datos, o la solución más práctica: almacenar los datos en una Base de Datos. Si bien, las mejoras podrían ser varias, la práctica me ayudó a comprender cómo se comportan las conexiones en red usando un modelo de bajo nivel como sockets y, más aún, a apreciar las facilidades que ofrecen tecnologías más modernas que abstraen esta complejidad

pero que nunca está de más entender para un futuro dónde se nos requiera entender el modelo para implementaciones más complejas.