Sistema de comunicación publicación/subscripción con RMI

Luis Jafet Ramos Castillo

4 de Marzo de 2016

1. Introducción

Los sistemas distribuidos pretenden realizar cómputo que en muchas ocasiones requiere de la comunicación entre procesos alojados en distintos lugares. Principalmente, existen dos paradigmas: llamada a procedimientos remotos (RPC) e invocación de métodos remotos (RMI).

RMI es un modelo equivalente a RPC, pero orientado a objetos sobre aplicaciones distribuidas. Los objetos distribuidos dentro de una red proporcionan métodos que dan accesos a distintos servicios. Se tiene un cliente activo que invoca los métodos del servidor y este los devuelve al cliente.

Los responsables de convertir los parámetros de la aplicación cliente/servidor durante la la invocación remota es el suplente, mientras que las interfaces sirven para especificar el formato de los métodos remotos y es compartida por el cliente y el servidor. Asimismo, el enlace es la asociación entre el cliente y el servidor.

La comunicación publicación/subscripción es un sistema distribuido basado en eventos publicados por editores, mientras que los subscriptores expresan interés por un tipo particular de eventos. Un evento se envía a un conjunto de subscriptores y existen distintos esquemas de filtro, se destacan los siguientes dos

- Basados en canales: los eventos se publican a un canal y se puede subscribir a este para publicar o recibir dichos eventos.
- Basados en contenido: se realiza una consulta sobre los valores de los atributos del evento.

2. Objetivo

Con base en el modelo RMI, se pretende implementar un sistema de publicación/subscripción que satisfaga los esquemas anteriormente mencionados. En este proyecto, los mensajes representan los eventos publicados, de esta manera en el esquema basado en contenido se busca que el cuerpo del texto contenga algo en específico. También se desea que los clientes pueda recuperar el historial de mensajes publicados en sus respectivas subscripciones.

3. Implementación

A continuación, se muestra la implementación del sistema en *Java* con el código principal acompañado de su explicación y de porque se utilizaron ciertas estructuras o recursos. Es importante mencionar que los mensajes escritos en la terminal o en los archivos se acortaron, pero en la siguiente dirección, se encuentra el código completo `https://github.com/luisjafet/Distribuido/tree/master/proyecto1`.

Los mensajes contienen el nombre del cliente que lo publicó, el texto y canal asociados a este, además de una variable que indica si el esquema es basado en contenido.

```
1 public Message(String name, String channel, String text, boolean
      content) {
2     this.name = name;
3     this.text = text;
4     this.channel = channel;
5     this.content = content;
6 }
```

Listing 1: constructor de la clase Mensaje

Las interfaces para el cliente y servidor proveen métodos para el manejo su subscripción, cancelación de esta y envió/recepción de mensajes.

```
1 public interface IChatClient extends java.rmi.Remote {
2     void receiveEnter(String name, String channel) throws
    RemoteException;
3     void receiveExit(String name, String channel) throws
    RemoteException;
4     void receiveMessage(Message message) throws RemoteException;
5 }
6 public interface IChatServer extends java.rmi.Remote {
7     void login(String name, String channel, IChatClient
    newClient) throws RemoteException;
8     void logout(String name, String channel) throws
    RemoteException;
9     void send(Message message) throws RemoteException;
10 }
```

Listing 2: constructores de las interfaces Cliente y Servidor

El cliente requiere un nombre, la dirección del servidor al que se conectará, una variable booleana que indique si será un esquema basado en contenido y el canal al que inicialmente estará subscrito. Es importante destacar que si se tiene un esquema basado en contenido la variable canal guarda el nombre del contenido al que se está subscribiendo el cliente.

Por último, se tiene un *ArrayList* que almacena el nombre de los canales o contenidos a los que el cliente esta subscrito. De esta manera, un cliente puede tener varias subscripciones determinados por el *heap* en lugar de fijar un valor en el código, dado que no se tiene que indicar un tamaño inicial en la estructura.

Se implementó de esta manera y no con un conjunto, ya que posteriormente se requerirá iterar sobre la estructura de datos y el *ArrayList* provee una solución natural para ello.

```
1 public ChatClient(String name, String channel, String url,
    boolean content) throws RemoteException {
2     this.name = name;
3     serverURL = url;
4     this.content = content;
5     channels = new ArrayList<String>();
6     login(channel);
7 }
```

Listing 3: constructor de la clase Cliente

Por un lado, el esquema basado en contenido se implementó en el método de recepción del cliente, ya que solo si el contenido deseado se encuentra en el texto del mensaje, entonces se entregará el mensaje.

Por otro lado, el esquema basado en canales se implementó de tal manera que se entreguen los mensajes si el atributo canal del mensaje se encontraba en el *ArrayList* del cliente, que es el que almacena sus subscripciones.

```
1  public void receiveMessage(Message message) {
2      if (content) {
3          for (int i = 0; i < channels.size(); i++) {
4              if (message.text.toLowerCase().contains(channels.get
       (i))) {
5                  System.out.println(message.text + "\n");
6                  break;
7              }
8          }
9      } else if (channels.contains(message.channel)) {
10         System.out.println(message.text + "\n");
11     }
12 }
```

Listing 4: método de recepción de mensajes

Por último, se muestra una parte del código del método en envió de mensajes que tiene el servidor en el que se van almacenando los mensajes en un archivo si pertenecen a un canal o contenido en específico (implementando el filtro como en el método anterior) y se envian los contenidos del archivo cuando el servidor recibe la petición *all*.

Se decidio utilizar archivos para no estar sujetos al *heap*, ya que es probable que la cantidad de mensajes sea mucho mayor a la de subscripciones (canales o contenidos).

```
if (message.text.equals("all")) {
    reader = new FileReader(message.channel);
    BufferedReader content = new BufferedReader(reader);
    while ((textRead = content.readLine()) != null) {
        textSend.append(textRead + "\n");
    }
    message.text = textSend.toString();
} else {
    writer = new FileWriter(message.channel, true);
    if (message.content) {
        if (message.text.toLowerCase().contains(message.channel)
    ) {
            writer.write("message.text + "\n");
        }
    } else {
        writer.write("message.text + "\n");
    }
    writer.close();
}
((IChatClient) entChater.nextElement()).receiveMessage(message);
```

Listing 5: método de recepción de mensajes

4. Conclusión

En conclusión, es posible implementar un sistema de publicación/subscripción con base en el modelo RMI. Asimismo, se pueden utilizar satisfactoriamente distintos esquemas, por ejemplo el basado en canales y contenido y el manejo de archivos facilita guardar los historiales de los canales y contenidos para ser entregados a los clientes.

Referencias

[1] Wan Fokkink. *Distributed Algorithms: An Intuitive Approach*. MIT Press, 2013.

[2] Sukumar Ghosh. *Distributed systems: an algorithmic approach*. CRC press, 2014.