Manual del laboratori de "Programació Concurrent i Distribuïda" JAVA Concurrent

Xavier Messeguer

March 26, 2008

Sessió

Comunicacions

En aquest capítol veurem com es poden comunicar dos programes executats en diferents sistemes.

Entenem un port (`socket`) com un punt de comunicació entre sistemes que correspon a un dispositiu físic d'entrada/sortida de la màquina. Java admet dos tipus de ports:

- Sockets Stream (TCP, Transmission Control Protocol): estan orientats a la connexió i les dades es transfereixen sense enquadrar-les en paquets.
- Sockets Datagrama (UDP, User Datagram Protocol): es un servei de transferència sense connexió. Les dades s'envien i es reben en paquets. Es poden perdre, arribar duplicats o en ordre diferent al de lliurament.

Nosaltres desenvoluparem comunicacions via TCP sota el protocol IP (Internet Protocol).

1.1 Aplicacions servidor/client

Les comunicacions en Java es dissenyen com aplicacions servidor/client. La idea general és que el servidor es posa a escoltar per un port i el client ha de demanar la comunicació. Desenvoluparem l'exemple d'un `Emisor` que envia paraules al `Receptor` que les escriu; el `Receptor` farà de servidor i l'`Emisor` de client.

En una aplicació servidor/client el servidor crea un *server socket*, és a dir, avisa al sistema en quin port estarà pendent:

```
ServerSocket ss = null;
⋮
ss = new ServerSocket(15000);
```

Utilitzará el port 15000 (entre 5000 i 65535) per comunicar-se amb altres programes ja que els anteriors estan reservats al sistema (podeu comprovar-ho mirant el fitxer */etc/services* de Unix). Posteriorment crea un canal de comunicació per cada client de la cua de peticions (instrucció `accept`) o es queda a l'espera si la cua està buida:

```
Socket sk = null;
⋮
sk=ss.accept();
```

El canal creat (`socket`) li servirà per comunicar-se amb el client. Seguidament, i donat que el servidor rebrà dades, associa un fluxe de entrada de dades (*InputStream*) a aquest canal

```
InputStream skin;
⋮
skin = sk.getInputStream();
```

i un filtre (*DataInputStream*) que permet transformar la codificació de les dades. En el exemple rep en `UTF` (*Unicode Transfer Format*) i el transforma en un `String`:

```
DataInputStream dis;
⋮
dis = new DataInputStream (skin);
⋮
...  dis.readUTF();
```

Aquí només hem creat un fluxe d'entrada de dades, però els canals ofereixen la possibilitat de crear-ne dos: un d'entrada i un de sortida.

El codi complet del servidor és:

```
importjava.io.*;
import java.net.*;

public class ReceptorParaules {
   public static void main(String[] args) throws IOException {
      ServerSocket ss = null;
      Socket sk = null;
      InputStream skin;
      DataInputStream dis;

      try {
        ss = new ServerSocket(9500);
        sk=ss.accept();
        skin = sk.getInputStream();
        dis = new DataInputStream(skin);
           // Llegeixo entrada pel canal
        String st = new String (dis.readUTF());
        while (st.length()>0){
           System.out.print(st);
           st = dis.readUTF();
        }
        dis.close();
        skin.close();
        sk.close();
      catch (IOException e) { }
   }
}
```

Observeu que el servidor anirà llegint les paraules que rep pel canal de comunicació i les anirà escrivint per pantalla.

El client del nostre exemple es vol connectar amb el servidor per enviar-li paraules, i ho fa especificant l'adreça IP i el port en el què el servidor està escoltant:

```
Socket sk = null;
⋮
sk = new Socket("127.0.0.1",15000);
```

Si el servidor no ha obert el socket es produeix una excepció d'entrada/sortida. L'adreça del propi ordinador és `127.0.0.1`, però si algu vol connectar-se a nosaltres cal esbrinar l'adreça real obrint una finestra `MSDOS` i fent `winipcfg`. Si treballen en Linux cal fer `hostname` per saber el nom i `dig` per saber l'IP.

Un cop obert el socket, associem un fluxe de sortida (*OutputStream*) i un filtre (*DataOutputStream*) que servirà per transformar un `String` a `UTF`:

```
OutputStream skout;
DataOutputStream dos;
⋮
skout = sk.getOutputStream();
dos = new DataOutputStream(skout);
⋮
dos.writeUTF(miss);
```

El codi complet del client, que envia repetidament la mateixa paraula, és:

```
importjava.io.*;
import java.net.*;

public class EmisorParaules {
   public static void main(String[] args) throws IOException {
      Socket sk = null;
      OutputStream skout;
      DataOutputStream dos;

      try {
        sk=new Socket("127.0.0.1",15000);
        skout = sk.getOutputStream();
        dos = new DataOutputStream (skout);
        String miss="Hola";
        for (int i=0;i<100;i++)
           dos.writeUTF(miss);
        dos.writeUTF("");
        dos.close();
        skout.close();
        sk.close();
      } catch (IOException e) { }
   }
}
```

1.2 Servidors multifil

En aquesta secció veurem com un servidor pot donar servei a molts clients independents entre ells, és a dir, asíncrona. La idea és que per a cada client es crei un canal de comunicació atés per un fil creat pel servidor:

```
import java.net.*;
import java.io.*;
public class ReceptorParaulesMultiFil {
   public static void main(String[] args) throws IOException {
      ServerSocket ss = null;
      ss = new ServerSocket(15000);
      while (1==1){
         Socket sk = null;
         sk = ss.accept();
         ThreadServidorThread t = new ReceptorParaulesThread(sk);
         t.start();
      }     }
}
```

I el codi del fil servidor que atendria les entrades de cada client:

```
import java.net.*;
import java.io.*;
public class ReceptorParaulesThread extends Thread{
   Socket sk = null;
   public ReceptorParaulesThread (Socket SK){
      sk=SK;
   }
   public void run(){
      try {
         InputStream skin = sk.getInputStream();
         DataInputStream dis = new DataInputStream(skin);
         String s = new String(dis.readUTF());
         ⋮
         while (⋮){
            ⋮
            s=dis.readUTF();
         }
         dis.close();
         skin.close();
         sk.close();
      } catch (IOException e)
   }
}
```

1.3 Lectors i escriptors

Aquest títol modelitza l'accés concurrent a una Base de Dades d'usuaris que poden llegir (lectors) i modificar (escriptors) el contingut de la BD sota la següent condició: molts lectors poden consultar la BD simultàneament però els escriptors ho han de fer en exclusió mútua.

Per monitoritzar aquest comportament declarem una interfície amb quatre mètodes:

```
public interface ReadWrite {
   public void acquireRead() throws InterruptedException;
   public void releaseRead();
   public void acquireWrite() throws InterruptedException;
   public void releaseWrite();
}
```

Ara cal dissenyar una implementació justa i equilibrada dels dos tipus d'usuaris. Proposem una primer implementació:

```
public class ReadWriteSafe implements ReadWrite {
   private boolean writing = false;
   private int readers = 0;

   public synchronized void acquireRead() throws InterruptedException {
      while (writing) wait();
      ++readers;
   }
   public synchronized void releaseRead() {
      --readers;
      if (readers == 0) notify();
   }
   public synchronized void acquireWrite() throws InterruptedException {
      while (readers > 0 || writing) wait();
      writing = true;
   }
   public synchronized void releaseWrite() {
      writing = false;
      notifyAll();
   }
}
```

Quins problemes planteja ... que intentem solucionar amb la segona implementació:

```
public class ReadWritePriority implements ReadWrite {
   private boolean writing = false;
   private int readers = 0;
   private int waitingW = 0;

   public synchronized void acquireRead() throws InterruptedException {
      while (writing || waitingW != 0) wait();
      ++readers;
   }
   public synchronized void releaseRead() {
      --readers;
      if (readers == 0) notify();
   }
   public synchronized void acquireWrite() throws InterruptedException {
      ++waitingW;
      while (readers > 0 || writing) wait();
      --waitingW;
      writing = true;
   }
   public synchronized void releaseWrite() {
      writing = false;
      notifyAll();
   }
}
```

Aquesta implementació també planteja un nou problema ... que solventem fent

```
public class ReadWriteFair implements ReadWrite {
   private boolean writing = false;
   private int readers = 0;
   private int waitingW = 0;
   private boolean readersturn = false;

   public synchronized void acquireRead() throws InterruptedException {
      while (writing || (waitingW != 0 && !readersturn)) wait();
      ++readers;
   }
   public synchronized void releaseRead() {
      --readers;
      readersturn=false;
      if (readers == 0) notifyAll();
   }
   public synchronized void acquireWrite() throws InterruptedException {
      ++waitingW;
      while (readers > 0 || writing) wait();
      --waitingW;
      writing = true;
   }
   public synchronized void releaseWrite() {
      writing = false;
      readersturn=true;
      notifyAll();
```

```
        }
    }
```

1.4 Exercicis

1. Suposeu que en un *host* teniu una BD (taula d'enters) en la qual els usuaris s'hi connecten via *sockets* lectors escriptors. Per monitoritzar la concurrència utilitzem els monitors

   (a) `ReadWriteSafe.java` que permet l'accés concurrent de lectors però que pot no permetre l'entrada d'escriptors

   (b) `ReadWriteFair.java` que permet l'accés concurrent de lectors i en exclusió mútua d'escriptors de forma justa i equilibrada.

   Implementeu aquesta BD i simuleu l'accés concurrent de desenes de lectors i escriptors via *sockets* (implementeu lectors i escriptors com *threads* decidint de forma aleatòria si són lectors o escriptors i la posició on llegeixen o escriuen).

   La sortida per pantalla ha de mostrar l'estat en que es troben els usuaris (en espera, llegint o escrivint, acabat, ...) i ha de permetre comprobar el funcionament dels dos monitors.

2. Dissenyeu un servidor multi-clients que sumi tots els nombres que li vagin enviant els clients.

3. Dissenyeu un servidor multi-clients tal que capgiri les frases que li envii cada client.