SYSTEMES D'INFORMATION 2

TP 1 - SOCKETS

Jean-Christophe ROLDAN & Samuel MERTENAT

Télécommunications
T3a & T3f

Fribourg, le 28 septembre 2015

TABLE DES MATIERES

SYSTEMES D'INFORMATION 2

TP 1 - SOCKETS

1 INTRODUCTION

Ce premier travail pratique consiste à développer une application élémentaire pour la communication interprocessus en utilisant des sockets TCP ou UDP, réalisée en Java.

2 QUESTIONS

2.1 APPLICATION CLIENT/SERVEUR TCP

2.1.1 P0

Indiquer les systèmes d'exploitation utilisés et les versions Java installées des deux côtés (client et serveur).

PC fixes : Windows 7 Entreprise 64 bits, Java 1.7 (Eclipse), Java 1.8

PC portable : Mac OS X El Capitan (GM), 64 bits, Java 1.8 (Eclipse), Java 1.8

2.1.2 P1

À quel moment exactement (c.-à-d. à l'exécution de quelle méthode) la connexion TCP est-elle établit ? Répondre pour les deux côtés, client et serveur.

Afin de déterminer le moment auquel la connexion est établie, nous utilisons le mode « debug » offert par Eclipse, en exécutant le programme étape par étape. Ainsi, nous pouvons constater que l'appel des deux méthodes ci-dessous initie la connexion entre le client et le serveur :

- Client : Lorsque l'utilisateur appuie sur le bouton « Connect », qui appelle à l'instanciation d'un « Socket »

```
s = new Socket(isa.getAddress(), isa.getPort());
```

- Serveur : Lorsque la connexion est initiée par le client

```
Socket s = serverSocket.accept();
```

2.1.3 P2

Est-ce que c'est nécessaire de spécifier le numéro de port TCP local côté client ? Si oui, comment peut-on le faire ? Si non, comment est-ce qu'on le ferait si on voulait quand-même le faire ?

Il n'est pas nécessaire de spécifier le numéro de port TCP local car la classe « Socket » propose plusieurs constructeurs ne nécessitant pas cette information, en employant un port attribué de manière aléatoire par le système d'exploitation (> 1023). Cependant, il est possible d'attribuer un port au client, en utilisant un de ces deux constructeurs :

```
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)
Creates a socket and connects it to the specified remote address on the specified
remote port.
```

Ou

```
Socket(String host, int port, InetAddress localAddr, int localPort)
Creates a socket and connects it to the specified remote host on the specified
remote port.
```

Source : http://docs.oracle.com/javase/7/docs/api/ → java.net → Class Socket

2.1.4 P3

Quel est l'effet de la fermeture du `ServerSocket` (évidemment sans fermer l'application), notamment par rapport aux connexions existantes ?

En consultant la documentation de l'API Java 1.7, nous pouvons constater que l'appel de la méthode « close() » de la classe « ServerSocket » entraine la fermeture du « Socket », mais aussi :

- Une exception (« SocketException ») si le thread est actuellement en train d'exécuter la méthode « accept() »
- La fermeture des canaux associés au « Socket »

```
public void close()
           throws IOException
Closes this socket. Any thread currently blocked in accept() will throw a
SocketException.
If this socket has an associated channel then the channel is closed as well.
Specified by:
close in interface Closeable
Specified by:
close in interface AutoCloseable
Throws:
IOException - if an I/O error occurs when closing the socket.
```

Donc, si la méthode « close() » est appelée, aucune nouvelle connexion ne pourra être initiée et les connexions actuelles se feront fermées.

Source : http://docs.oracle.com/javase/7/docs/api/ → java.net → Class ServerSocket

2.1.5 P4

Faire une mesure contenant l'établissement, l'envoi d'un seul message et la fermeture de la connexion. Documenter (diagramme en flèche) et expliquer les trames entre l'établissement et la fermeture de la connexion.

Afin de filtrer au mieux les paquets échangés lors de la capture, nous utilisons le filtre suivant : « tcp && ip.src==160.98.20.185 || ip.dst==160.98.20.185 » qui nous permet de récupérer uniquement les paquets de type TCP, envoyés ou reçus par le client (160.98.20.185). Nous obtenons ceci :

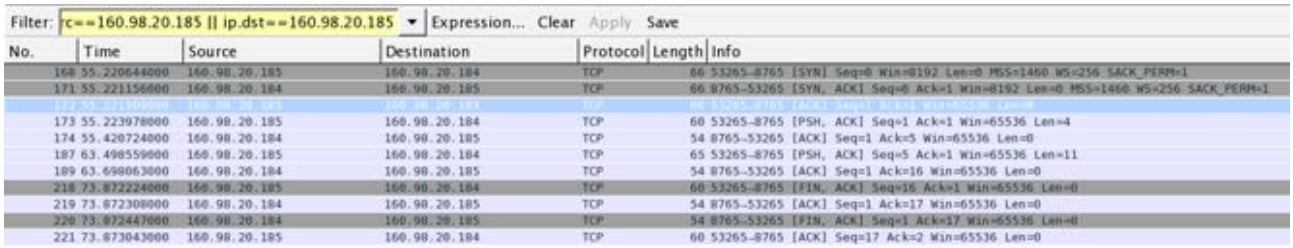

Figure 1: Liste des paquets échangés entre le client et le serveur

Ce qui nous donne, sous la forme d'un diagramme en flèches :

Dans cet échange, trois phases se distinguent ; nous avons tout d'abord l'établissement de la connexion, puis l'envoi des données et pour finir, la fermeture de la connexion. Les trois premières trames TCP échangées permettent l'établissement de la connexion entre le client et le serveur, initialisée du côté client par l'instanciation du « Socket ». Les quatre trames suivantes permettent l'envoi de données. Le client envoie un message, séparé en deux trames, qui sont acquittées immédiatement par le serveur par un acquittement (« ACK »). Les dernières trames permettent au client et au serveur de fermer, l'un après l'autre, leur connexion respective.

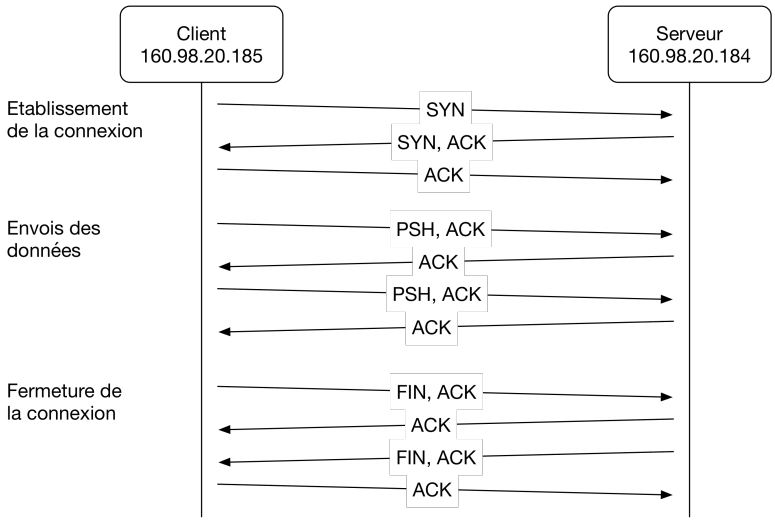

Figure 2: Diagramme en flèche d'une connexion TCP

Pour l'envoi du message « Hello 2 », deux trames (« PSH, ACK ») sont nécessaires, transportant des données de respectivement 4 et 11 octets.

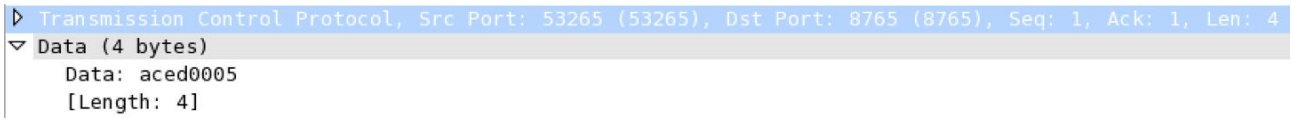

Figure 3: Première trame TCP de données

Les données débutent par la valeur « 0xaced », qui correspond à une constante définie dans l'API :

```
final static short STREAM_MAGIC = (short)0xaced;
```

Cette constante est employée à des fins de sérialisation et est suivie par un numéro de version, la constante « STREAM_MAGIC ».

```
static final short STREAM_MAGIC
Magic number that is written to the stream header.

static final short STREAM_VERSION
Version number that is written to the stream header.
```

La seconde trame, quant à elle, contient le message du client.

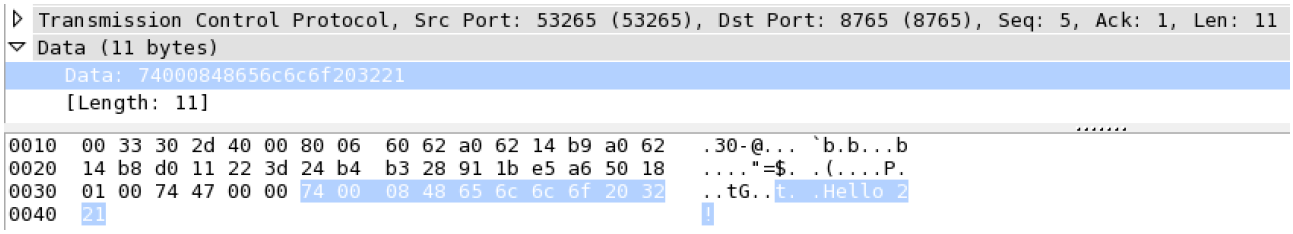

Figure 4: Seconde trame TCP de données

Source : http://docs.oracle.com/javase/7/docs/api/java/io/ObjectStreamConstants.html

2.1.6 P5

Documenter et expliquer le format (de la charge utile TCP) utilisé pour l'envoi du message vers le serveur.

Le message est envoyé sous la forme de « String ». En effet, les données émises lors de la seconde trame débutent par « 0x74 » (cf. figure ci-dessus).

```
final static byte TC_STRING = (byte)0x74;
```

Source : http://docs.oracle.com/javase/8/docs/platform/serialization/spec/protocol.html

2.1.7 P6

(a) Que se passe-t-il si l'application client envoi des messages avant que l'application serveur exécute la méthode `accept()` ?

En plaçant un « breakpoint » à la hauteur de la méthode « accept() », rien ne se passe sur la console lors de la connexion et de l'envoi d'un message de la part du client. Cependant, dès lors que nous reprenons l'exécution du programme côté serveur (exécution de la méthode « accept() »), celui-ci nous notifie de la connexion d'un client et nous transmet le message envoyé précédemment.

(b) Repérer et expliquer les tailles des fenêtres TCP lors de l'envoi consécutifs d'une dizaine de messages volumineux (toujours avant que le serveur exécute la méthode `accept()`). Documenter avec un diagramme en flèche qui donne une synthèse de l'échange.

Pour réaliser cette question, nous avons défini un buffer de 512 bytes à l'aide de la méthode « setReceiveBufferSize() » et avons émis 10 messages (« Msg1 » à « Msg10 »). En réalisant une capture Wireshark, nous obtenons les trames suivantes :

No.	Time	Source	Destination	Protocol	Length	Info
2	0.000027000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=5 Win=40829 Len=0 TSval=907745836 TSecr=907745836
3	6.403615000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=5 Ack=1 Win=12759 Len=7 TSval=907752236 TSecr=907745836
4	6.403656000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=12 Win=40828 Len=0 TSval=907752236 TSecr=907752236
5	10.250122000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=12 Ack=1 Win=12759 Len=7 TSval=907756077 TSecr=907752236
6	10.250173000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=19 Win=40827 Len=0 TSval=907756077 TSecr=907756077
7	14.368123000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=19 Ack=1 Win=12759 Len=7 TSval=907760182 TSecr=907756077
8	14.368173000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=26 Win=40826 Len=0 TSval=907760182 TSecr=907760182
9	18.923743000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=26 Ack=1 Win=12759 Len=7 TSval=907764733 TSecr=907760182
10	18.923869000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=33 Win=40826 Len=0 TSval=907764733 TSecr=907764733
11	22.649516000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=33 Ack=1 Win=12759 Len=7 TSval=907768457 TSecr=907764733
12	22.649567000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=40 Win=40825 Len=0 TSval=907768457 TSecr=907768457
13	26.317393000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=40 Ack=1 Win=12759 Len=7 TSval=907772114 TSecr=907768457
14	26.317445000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=47 Win=40824 Len=0 TSval=907772114 TSecr=907772114
15	30.145423000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=47 Ack=1 Win=12759 Len=7 TSval=907775941 TSecr=907772114
16	30.145472000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=54 Win=40823 Len=0 TSval=907775941 TSecr=907775941
17	33.613290000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=54 Ack=1 Win=12759 Len=7 TSval=907779404 TSecr=907775941
18	33.613341000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=61 Win=40822 Len=0 TSval=907779404 TSecr=907779404
19	37.798975000	127.0.0.1	127.0.0.1	TCP	63	54718→8765 [PSH, ACK] Seq=61 Ack=1 Win=12759 Len=7 TSval=907783586 TSecr=907779404
20	37.799025000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=68 Win=40821 Len=0 TSval=907783586 TSecr=907783586
21	41.940826000	127.0.0.1	127.0.0.1	TCP	64	54718→8765 [PSH, ACK] Seq=68 Ack=1 Win=12759 Len=8 TSval=907787723 TSecr=907783586
22	41.940876000	127.0.0.1	127.0.0.1	TCP	56	8765→54718 [ACK] Seq=1 Ack=76 Win=40820 Len=0 TSval=907787723 TSecr=907787723

Figure 5: Liste des trames émises pour l'envoi de 10 messages

Ces trames, mises dans un diagramme en flèches, nous permettent de bien pouvoir observer le fonctionnement des numéros de séquence, des acquittements, ainsi que du mécanisme de la fenêtre de réception du serveur. Comme nous pouvons l'observer, le serveur, en acquittant la trame réceptionnée, indique au client la valeur de sa fenêtre de réception, décrémentée à chaque fois de 1.

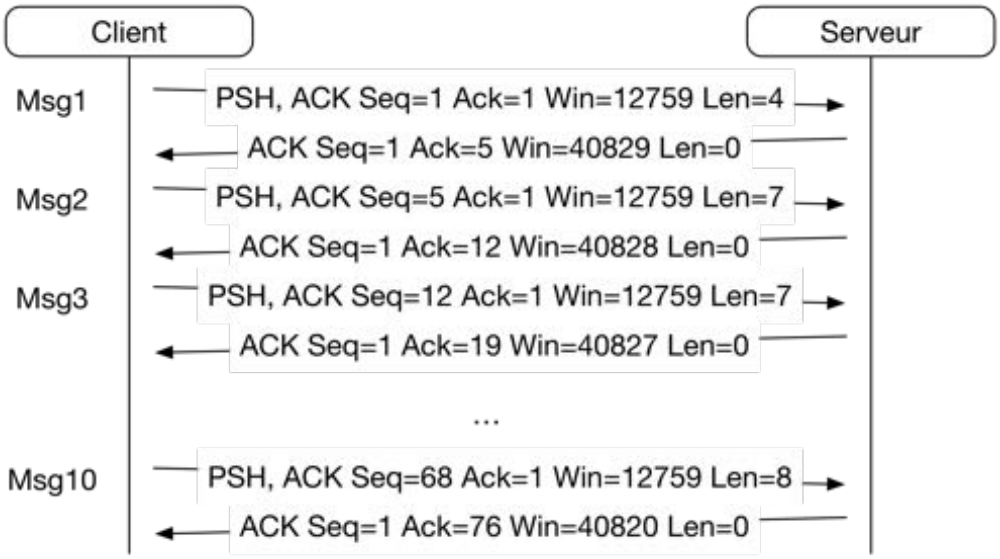

Figure 6: Numéros de séquence, d'acquittement et taille de la fenêtre TCP lors de l'envoi de données

(c) Quel est l'utilité de la méthode `setReceiveBufferSize()` dans ce contexte ?

D'après la documentation de l'API, la méthode « setReceiveBufferSize » permet de définir la taille des données pouvant être bufferisées tant que la méthode « accept() » n'a pas été appelée. Elle permet aussi de définir la taille de la fenêtre TCP.

```
setReceiveBufferSize
public void setReceiveBufferSize(int size)
                          throws SocketException
Sets the SO_RCVBUF option to the specified value for this Socket. The SO_RCVBUF
option is used by the platform's networking code as a hint for the size to set the
underlying network I/O buffers.
Increasing the receive buffer size can increase the performance of network I/O for
high-volume connection, while decreasing it can help reduce the backlog of incoming
data.
```

Source : http://download.java.net/jdk7/archive/b123/docs/api/java/net/Socket.html

2.2 APPLICATION CLIENT/SERVEUR UDP

2.2.1 P8

Retrouver le format du message et vérifier le contenu et les ports UDP utilisés.

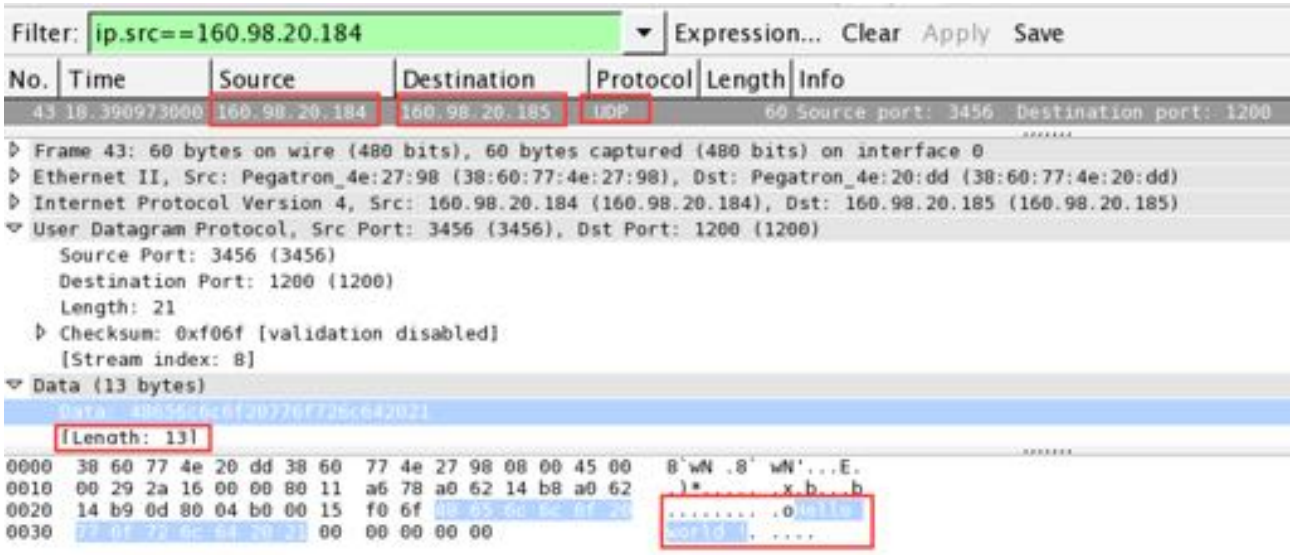

Figure 7: Aperçu du contenu d'un paquet UDP

Lorsque que le client envoie un message à destination du serveur, un paquet UDP est transmis. Sur la figure ci-dessous, nous pouvons observer que le port de source est un port attribué de manière aléatoire (3456) ; quant au port de destination, il correspond au port configuré par le serveur (1200). Le message est transmis en clair, sous la forme de caractères et correspond au nombre de bytes indiqué par le champ « Data » / « Length » du paquet UDP (13 caractères, « Hello world ! »).

2.2.2 P9

Que se passe-t-il si le serveur n'est pas fonctionnel (port serveur pas ouvert) ? Est-ce que le client peut le remarquer ?

Lorsque le serveur n'est pas démarré ou que le client n'utilise pas la bonne adresse / le bon port, il se fait notifier d'un paquet ICMP. Comme nous pouvons l'observer sur la capture ci-dessous, le paquet indique au client « Destination unreachable (Port unreachable) » (ICMP, type 3, code 3).

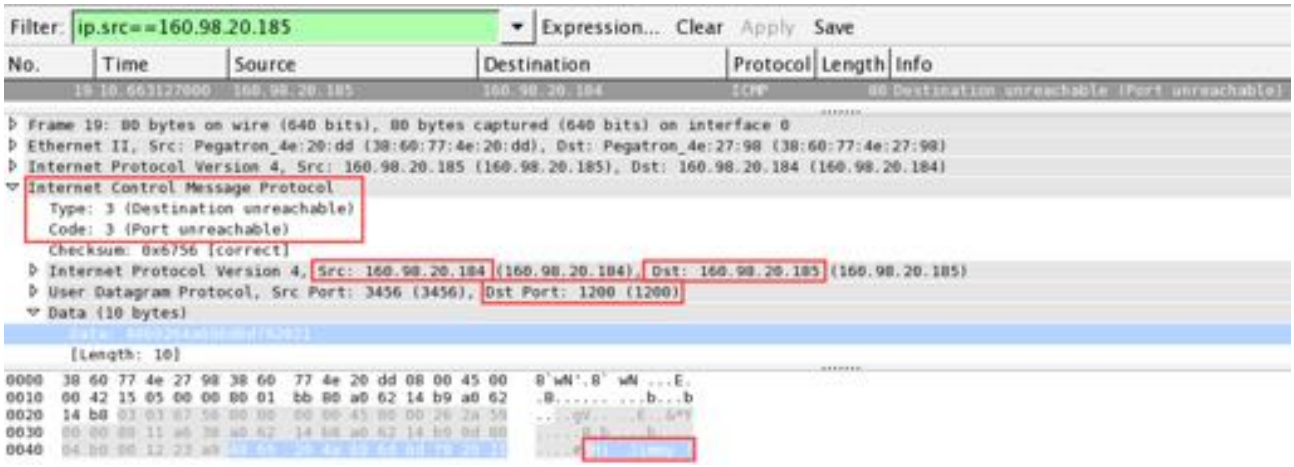

Figure 8: Envoi d'un paquet UDP lorsque le port de destination n'est pas atteignable

2.2.3 P10

Que se passe-t-il si l'émetteur émet son message avant que le récepteur appelle sa méthode `receive()` ?

Tant que le « DatagramSocket » n'est pas instancié, les messages (« DatagramPacket ») ne pourront pas être récupérés. Cependant, une fois instancié, même si la méthode « receive() » n'a point encore été appelée, les messages peuvent être bufferisés et lus à son appel.

3 CONCLUSION

Ce premier travail pratique a été l'occasion de comprendre le fonctionnement des Sockets et de réaliser une implémentation client / serveur, en se basant respectivement sur les protocoles UDP et TCP.

4 ANNEXE

4.1 TCP

4.1.1 SERVERTHREAD.JAVA

```
package sockets.tcp;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.net.Socket;

public class ServerThread extends Thread {

  private Socket socket;
  private int    id;
```

```
  public ServerThread(Socket s, int id) {
    this.socket = s;
    this.id = id;
  }

  public void run() {
    System.out.println(">> " + id + " : New connection request
from "
        + socket.getInetAddress() + ":" + socket.getPort());
    try {
      ObjectInputStream data = new
ObjectInputStream(socket.getInputStream());
      while (true) {
        String msg = (String) data.readObject();
        System.out.println("@@ " + id + " : " + msg);
      }
    } catch (IOException | ClassNotFoundException e1) {
      try {
        socket.close();
        System.out.println("## " + id + " : connection closed");
      } catch (Exception e2) {
      }
    }
  }

}
```

4.1.2 TCPCLIENT.JAVA

```
package sockets.tcp;

import java.net.*;
import java.io.*;

public class TCPClient {
  private Socket             s;
  private ObjectOutputStream oos;

  public TCPClient(InetSocketAddress isa) {
    try {
      s = new Socket(isa.getAddress(), isa.getPort());
      oos = new ObjectOutputStream(s.getOutputStream());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void closeSocket() {
    try {
      s.close();
    } catch (IOException e) {
      e.printStackTrace();
```

```
    }
  }

  public void sendMsg(String msg) {
    try {
      oos.writeObject(msg);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

}
```

4.1.3 TCPSERVER.JAVA

```
package sockets.tcp;

import java.net.*;
import java.io.*;

public class TCPServer {

  static final int    SERVER_PORT = 8765; // server port to use
  static int          id          = 0;          // client id
  static ServerSocket serverSocket;

  public static void main(String[] args) {
    try {
      serverSocket = new ServerSocket(SERVER_PORT);
      serverSocket.setReceiveBufferSize(512);
      while (true) {
        Socket s = serverSocket.accept();
        System.out.println("Listening on TCP port " +
s.getLocalPort() + "...");
        ServerThread serverThread = new ServerThread(s, id++);
        serverThread.start();
      }
    } catch (Exception e) {
    } finally {
      try {
        serverSocket.close();
      } catch (IOException e) {
      }
    }
  }

}
```

4.2 UDP

4.2.1 UDPCLIENT.JAVA

```
package sockets.udp;

import java.net.*;

public class UDPClient {

  DatagramSocket ds;
  DatagramPacket dp;
  byte[]         buffer;

  public UDPClient(int localport) throws SocketException {
    ds = new DatagramSocket(localport);
  }

  public void sendMsg(InetSocketAddress isaDest, String msg)
throws Exception {
    dp = new DatagramPacket(msg.getBytes(), msg.length(),
isaDest.getAddress(),
        isaDest.getPort());
    ds.send(dp);
  }

  public void closeSocket() {
    ds.close();
  }

  public int getPort() {
    return ds.getLocalPort();
  }

}
```

4.2.2 UDPSERVER.JAVA

```
package sockets.udp;

import java.io.IOException;
import java.net.*;

public class UDPServer {

  static final int       MAX_SIZE    = 100;
  static final int       SERVER_PORT = 1200; // port to use
  static DatagramSocket ds;

  public static void main(String[] args) {
    byte[] buffer = new byte[MAX_SIZE];
    DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
    int counter = 1;
```

```
    System.out.println("Serving UDP port " + SERVER_PORT + "...");

    try {
      ds = new DatagramSocket(SERVER_PORT);
      while (true) {
        dp.setLength(buffer.length);
        ds.receive(dp);

        String msg = new String(buffer, 0, dp.getLength());
        System.out.println("Message " + counter++ + " from : " +
dp.getAddress()
            + ":" + dp.getPort() + " -> " + msg);
      }
    } catch (SocketException e) {
      e.printStackTrace();
      ds.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

}
```