Java Socket Programming

November 3, 2017

The purpose of this worksheet is to teach the fundamentals of network programming in Java, in particular we cover UDP and TCP.

1 UDP and Datagrams

The simplest form of communication between applications running on separate machines on the net is to use UDP (User Datagram Protocol). In this section we develop two applications which send messages to each other using UDP.

UDP uses a combination of sockets and packets in the program to send short messages over the internet to machines addressed by a combination of their machine name (or IP address) and a port number. This is a connectionless protocol. Messages are sent off containing the address of their recipient and the underlying IP protocol arranges that they are delivered to their destination. Messages can be lost. We will try to illustrate this in the exercises.

Consider the following application.

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

class UDPSender{
  public static void main(String [] args){
    try{
      InetAddress address =
         InetAddress.getByName("koestler.ecs.soton.ac.uk");
      DatagramSocket socket = new DatagramSocket();
      for(int i=0;i<10;i++){
        byte[] buf = String.valueOf(i).getBytes();
        DatagramPacket packet =
           new DatagramPacket(buf, buf.length, address, 4321);
        socket.send(packet);
        System.out.println("send DatagramPacket "
          + new String(packet.getData()) + " "
          + packet.getAddress() + ":"
          + packet.getPort());
```

```
        Thread.sleep(2000);
      }
    }catch(Exception e){System.out.println("error");}
  }
}
```

This application is a client which is constructing and sending messages as UDP packets. Various classes from the java.net package have been used. Most significantly we have constructed a DatagramSocket and a DatagramPacket and then called socket.send(packet) to put our message out on the network. The DatagramPacket contains, as well as the message, its destination in the form of an InetAddress and a port number (here 4321).

The application sends 10 numbered messages and prints a copy of what it has sent. The application sleeps for 2 seconds between each message so that a human user can observe its behaviour.

The corresponding application which is to receive these messages is:

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

class UDPReceiver{
  public static void main(String [] args){
    try{
      DatagramSocket socket = new DatagramSocket(4321);
      byte[] buf = new byte[256];
      for(int i=0;i<10;i++){
        DatagramPacket packet = new DatagramPacket(buf, buf.length);
        socket.receive(packet);
        System.out.println("receive DatagramPacket "
          + (new String(packet.getData())).trim() + " "
          + packet.getAddress() + ":"
          + packet.getPort());
      }
    } catch(Exception e){System.out.println("error "+e);}
  }
}
```

Again we need a DatagramSocket and a DatagramPacket, this time so we can call socket.receive(packet). Note that the application listens on port 4321 and must be running on machine koestler.ecs.soton.ac.uk in order to collect the packets sent by our earlier application. As each packet is received its details are printed so we can see what we have.

Exercise 1. Compile and run the applications shown here. You will need to change the name of the machine in UDPSender to the one you are using to run UDPReceiver. You can run both applications on the same machine, but it is a

little more exciting if you use two machines. You will have to start the receiver first. It will wait for messages to arrive. Observe the port numbers assigned to the messages. When sending, the port number has been assigned by the client machine and will vary from run to run. If you move the DatagramSocket assignment inside the loop in UDPSender you will see that a different port is used for each send.

Exercise 2. Let us try to show message loss. The simplest way to do this is to start the sender first. By the time you start the receiver you will probably have missed some of the earlier messages.

Exercise 3. Let us try to show the kind of message loss which occurs when the receiver can't keep up with the sender. This is not easy, because UDP maintains quite large buffers. The changes suggested here will require some ingenuity on your part, because the demonstration will depend upon the power of your machines and the load on the network.

First step is to speed up the sender and slow down the receiver. Remove the Thread.sleep call from the sender altogether, so the sender will operate as fast a possible. Now add a Thread.sleep call to the receiver so that is wastes time, in order to increase the likelyhood of missing something. Start the receiver first. With just 10 messages sent, the receiver won't miss anything. The messages have been buffered on receipt. Message loss will only occur when this buffer overflows.

If you increase the number of messages sent and received then you might be able to demonstrate this overflow. It will be difficult to observe by eye, so you may wish to add some code to detect the loss.

Exercise 4. Try running one copy of the receiver application and two copies of the sender. Use the slow sender, so that you have time to start the second one before the first has finished. Note how the receiver receives messages from both in an arbitrary interleaving.

Exercise 5. Modify the sender and the receiver so that the receiver sends a reply to acknowledge the receipt of each message. Make sure that the sender waits for this acknowledgement before sending the next message. What effect does this have on the order in which the two applications can be started? Can you still run two senders and one receiver?

2 TCP and Streams

We can repeat the experiment of the previous section using Java's support for stream communication over TCP (Transmission Control Protocol).

Here is a simple TCP client, which sends a sequence of ten lines to a TCP Socket and then terminates.

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

class TCPSender{
  public static void main(String [] args){
    try{Socket socket = new Socket("koestler.ecs.soton.ac.uk",4322);
      PrintWriter out = new PrintWriter(socket.getOutputStream());
        for(int i=0;i<10;i++){
          out.println("TCP message "+i); out.flush();
          System.out.println("TCP message "+i+" sent");
          Thread.sleep(1000);
      }
    }catch(Exception e){System.out.println("error"+e);}
  }
}
```

The new concept here is the class Socket which can be used to set up stream communication between two applications. Here the client is trying to establish communication with the server koestler.ecs.soton.ac.uk on port 4322. If a connection is achieved, then the method socket.getOutputStream is used to get a stream which can be used to communicate with a server listening on the same port. Messages can now be sent to that stream. We use a PrintWriter to be able to send text messages a line at a time. Sockets have two-way communication, so we could have also got an input stream from this socket, but it wasn't required for this simple client.

The corresponding server is a little more interesting.

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

class TCPReceiver{
  public static void main(String [] args){
    try{
      ServerSocket ss = new ServerSocket(4322);
        for(;;){
          try{Socket client = ss.accept();
            BufferedReader in = new BufferedReader(
              new InputStreamReader(client.getInputStream()));
            String line;
            while((line = in.readLine()) != null)
              System.out.println(line+" received");
            client.close();
        }catch(Exception e){System.out.println("error "+e);}
      }
    }catch(Exception e){System.out.println("error "+e);}
  }
}
```

This time we need to set up our Socket for input in two stages. First we create a ServerSocket on a particular port, which listens for connections. We listen on 4322. The call Socket client = ss.accept() blocks and waits until a client attempts a connection. When that happens, a new port is allocated and a new socket opened on that port. In this application, that socket is assigned to the variable client.

We open a new input stream on that socket and the client can now talk to the server. It reads each line sent by the server, one at a time until client closes its socket, whereupon the server also closes its socket.

The server will then loop back and wait for another connection. The server is single-threaded, so it deals with each client sequentially. We will revise it to deal with each client concurrently in the next section, after we have played with this pair of applications for a while.

Exercise 6. Compile and run the applications shown here. You will need to change the name of the machine in TCPSender to the one you are using to run TCPReceiver. Run the receiver first, then run the sender. Observe. Try it with two senders. Note that the second doesn't connect until the first has finished, so the behaviour is different from UDP. Also try starting the sender before the receiver. You will get an informative error message.

Exercise 7. Observe the effect of deleting out.flush() from the sender.

Exercise 8. Note that the receiver can be left running on a machine, servicing the requests of a series of senders. The receiver will happily run for days without error. Why is this? What have we done in its design to ensure this?

3 A multi-threaded TCP server

It is the receiver which we need to alter in order to have it handle clients concurrently rather than in sequence. Here is a possible revision, where we have chosen to embed the new Thread required to create this concurrency, rather than separate its behaviour into a new class. Either solution would have sufficed. It is a matter of taste which is to be preferred. The (slightly longer) version which uses a new class is given later in this section.

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

class TCPReceiverThreaded{
  public static void main(String [] args){
    try{
      ServerSocket ss = new ServerSocket(4322);
      for(;;){
        try{final Socket client = ss.accept();
```

```
          new Thread(new Runnable(){
            public void run(){try{
              BufferedReader in = new BufferedReader(
              new InputStreamReader(client.getInputStream()));
              String line;
              while((line = in.readLine()) != null)
                System.out.println(line+" received");
              client.close(); }catch(Exception e){}
            }
          }).start();
        }catch(Exception e){System.out.println("error "+e);}
      }
    }catch(Exception e){System.out.println("error "+e);}
  }
}
```

What happens here is that, as soon as a client connects, a new Thread is constructed and started to service that client's request and the main for-loop cycles immediately back to the top to wait for another connection. The newly created Thread services the connected client. If a second client connects, another Thread is created. Many clients can connect and their servicing will be interleaved. The experiments suggested in the exercises will demonstrate this.

Before we start the exercises, here is the server again, this time using a nested class to construct the concurrent Threads. Some readers may prefer this version.

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

class TCPReceiverThreadedClass{
  public static void main(String [] args){
    try{
      ServerSocket ss = new ServerSocket(4322);
        for(;;){
          try{Socket client = ss.accept();
            new Thread(new ServiceThread(client)).start();
        }catch(Exception e){System.out.println("error "+e);}
      }
    }catch(Exception e){System.out.println("error "+e);}
  }

  static class ServiceThread implements Runnable{

    Socket client;

    ServiceThread(Socket c){client=c;}
```

```
    public void run(){try{
      BufferedReader in = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
      String line;
      while((line = in.readLine()) != null)
         System.out.println(line+" received");
      client.close(); }catch(Exception e){}
    }
  }
}
```

Exercise 9. Compile and run either of the server applications shown here with two or more of the clients from the previous section. Observer that the servicing is now interleaved.

Exercise 10. Extend the server application to print out details the machine which has connected. Adding +" from "+ client.getInetAddress() to the output should be sufficient, but you might want to do more. This information is more interesting of course if your clients are running on different machines.

Exercise 11. Extend the client and server applications so that the server acknowledges each line and the client waits for this acknowledgement before sending the next line.

4 A File Transfer Server and Client

This is a more elaborate example. We will not describe it in detail. The idea is that the reader experiment with these applications and modify them substantially. The basic server structure of the previous sections will be used to build a File Transfer server, which can be run on a remote machine and support transfer of files to and from that machine. The required client application will also be developed. The code given here is reasonably fault tolerant. It will transfer any files, not just text files. It is single-threaded. One of the exercises is to make a multi-threaded version. It uses TCP connections.

The Server is as follows.

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

public class FTServer {
  public static void main(String[] args) throws IOException {
    try{ServerSocket ss = new ServerSocket(4323);
      for(;;){
        try{
```

```
            System.out.println("waiting for connection");
            Socket client = ss.accept();
            System.out.println("connected");
            InputStream in = client.getInputStream();
            byte[] buf = new byte[1000]; int buflen;
            buflen=in.read(buf);
            String firstBuffer=new String(buf,0,buflen);
            int firstSpace=firstBuffer.indexOf(" ");
            String command=firstBuffer.substring(0,firstSpace);
            System.out.println("command "+command);
            if(command.equals("put")){
              int secondSpace=firstBuffer.indexOf(" ",firstSpace+1);
              String fileName=
                      firstBuffer.substring(firstSpace+1,secondSpace);
              System.out.println("fileName "+fileName);
              File outputFile = new File(fileName);
              FileOutputStream out = new FileOutputStream(outputFile);
              out.write(buf,secondSpace+1,buflen-secondSpace-1);
              while ((buflen=in.read(buf)) != -1){
                System.out.print("*");
                out.write(buf,0,buflen);
              }
              in.close(); client.close(); out.close();
            } else
            if(command.equals("get")){
              int secondSpace=firstBuffer.indexOf(" ",firstSpace+1);
              String fileName=
                    firstBuffer.substring(firstSpace+1,secondSpace);
              System.out.println("fileName "+fileName);
              File inputFile = new File(fileName);
              FileInputStream inf = new FileInputStream(inputFile);
              OutputStream out = client.getOutputStream();
              while ((buflen=inf.read(buf)) != -1){
                System.out.print("*");
                out.write(buf,0,buflen);
              }
              in.close(); inf.close(); client.close(); out.close();
            } else
            System.out.println("unrecognised command");
          } catch(Exception e){System.out.println("error "+e);}
        }
       }catch(Exception e){System.out.println("error "+e);}
     System.out.println();
    }
}
```

The basic structure of this server is the same as our TCPReceiver of the previous section. It waits for and accepts connections, this time on port 4323. Having accepted a connection, it reads the input stream as a series of buffers (as byte-arrays). It expects the first buffer to contain a command of the form either get filename or put filename. It parses this buffer, expecting the command (get or put) to be terminated by a space character and the filename to be terminated by a second space character. In the case of the put command, the bytes following the space following the filename will be the contents of a file being transferred to the server.

In the case of the get command, the server opens that file if it exists and begins to transmit the contents of the file over the output stream of the socket connected to the client. On completion of sending the file, the server reverts to waiting for a connection.

In the case of the put command, the server opens a file with that name for writing (whether it already exists or not). It then proceeds to copy the rest of the input stream from the socket to the file. On completion, the server reverts to waiting for a connection.

The corresponding client for this server is

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

public class FTClient {
  public static void main(String[] args) throws IOException {
    System.out.println(args[0]+" "+args[1]+" "+args[2]+" "+args[3]);
    if(args[1].equals("put")){
      File inputFile = new File(args[2]);
      FileInputStream in = new FileInputStream(inputFile);
      try{
        Socket socket = new Socket(args[0],4323);
        OutputStream out = socket.getOutputStream();
        out.write(("put"+" "+args[3]+" ").getBytes());
        byte[] buf = new byte[1000]; int buflen;
        while ((buflen=in.read(buf)) != -1){
          System.out.print("*");
          out.write(buf,0,buflen);
        }
        out.close();
      }catch(Exception e){System.out.println("error"+e);}
      System.out.println();
      in.close();
    } else
    if(args[1].equals("get")){
      File outputFile = new File(args[2]);
      FileOutputStream outf = new FileOutputStream(outputFile);
      try{
```

```
        Socket socket = new Socket(args[0],4323);
        OutputStream out = socket.getOutputStream();
        InputStream in = socket.getInputStream();
        out.write(("get"+" "+args[3]+" ").getBytes());
        byte[] buf = new byte[1000]; int buflen;
        while ((buflen=in.read(buf)) != -1){
          System.out.print("*");
          outf.write(buf,0,buflen);
        }
        out.close();
        in.close();
      }catch(Exception e){System.out.println("error"+e);}
      System.out.println();
      outf.close();
    } else
    System.out.println("unrecognised command");
  }
}
```

Again, the client has the same structure as the client from the previous section. It connects to a socket on port 4323 (which is where we hope the server is running). This client accepts commands on the command line, as follows

```
java FTClient remoteComputer put localFilename remoteFilename
java FTClient remoteComputer get localFilename remoteFilename
```

The put command to the client constructs a put command to be sent to the server. It connects to the server on remoteMachine then sends the command put remoteFilename followed by the contents of the file it finds locally at localFilename. This is the form that the server expects.

The get command to the client constructs a get command to be sent to the server. It then connects to the server on remoteMachine and sends that command. It reads the reply on the sockets input stream and stores this locally in the file localFilename.

Exercise 12. Compile and execute these applications, ideally on two or more machines. Observe that files are correctly transferred both by get and put.

Exercise 13. Make the server multithreaded, by allocating a separate thread to service each client.

Exercise 14. If the server is unable to locate a file which has been requested it reports the error at the server end but does not tell the client, which simply receives an empty file. Modify server and client to handle this situation more appropriately.

Exercise 15. Show that the system can transfer binary files correctly by putting a .class file onto the server and then executing it there.