PROJECT 2: RELIABLE DELIVERY

For this project we will explore the challenges associated with providing reliable delivery services over an unreliable network. Three different ARQ approaches have been described in class: stop-and-wait, concurrent logical channels and sliding window. Your task during this project is to develop a set of Java classes capable of reliably transferring a file between two hosts over UDP (TCP already offers reliable delivery services).

Students will be provided with a `UDPSocket` class that extends the `DatagramSocket` class provided by Java to send a `DatagramPacket` over UDP. Take time to familiarize yourself with the main two classes provided by Java for UDP support and their functionality. The `UDPSocket` class has the additional capability of allowing programmers to modify the MTU (to model fragmentation), the packet drop rate (to model unreliable networks) and add random time delays to packets (to model out-of-order reception).

Support material

Parameters regulating message transfers using the UDPSocket class are located in a file named unet.properties. This file may be modified to test different scenarios (the file includes comments describing each parameter). Note that you can also use this file to enable and configure logging options (to file or standard output stream).

Unet.properties

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# unet properties file
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#== log file name ================================
#
# log.filename = system (standard output stream)

log.filename = system
log.enable = true

#== packet drop rate =============================
#
#  n<0  --- Drop random packets (n*-100) probability
#  n=0  --- Do not drop any packets
#  n=-1 --- Drop all packets
#  n>0  --- Drop every nth packet (assuming no delay)
#  n=[d,]*d --- Drop select packets e.g. packet.droprate = 4,6,7

packet.droprate = 0

#== packet delay rate ============================
#
# packet delay in milliseconds
# min==max --- in order delivery

packet.delay.minimum = 10
packet.delay.maximum = 100

#== packet maximum transmission size =============
#
# -1 unlimited (Integer.MAX_VALUE)

packet.mtu = 1500
```

A brief tutorial on how to use UDP in Java is provided in the Appendix. A zip file (unet.zip) containing sample code, a supporting jar file (unet.jar) as well as the default unet.properties file will be provided.

NOTE: You may download all the support files from Harvey.

Programs

A minimum of two classes will have to be submitted, RSendUDP.java and RReceiveUDP.java (do not make them part of the edu.utulsa.unet package) where each class must implement the edu.utulsa.unet.RSendUDPI and edu.utulsa.unet.RReceiveUDPI interfaces respectively (as specified below, provided in unet.jar).

```
package edu.utulsa.unet;
import java.net.InetSocketAddress;
public interface RSendUDPI {
      public boolean setMode(int mode);
      public int getMode();
      public boolean setModeParameter(long n);
      public long getModeParameter();
      public void setFilename(String fname);
      public String getFilename();
      public boolean setTimeout(long timeout);
      public long getTimeout();
      public boolean setLocalPort(int port);
      public int getLocalPort();
      public boolean setReceiver(InetSocketAddress receiver);
      public InetSocketAddress getReceiver();
      public boolean sendFile();
}
package edu.utulsa.unet;
public interface RReceiveUDPI {
      public boolean setMode(int mode);
      public int getMode();
      public boolean setModeParameter(long n);
      public long getModeParameter();
      public void setFilename(String fname);
      public String getFilename();
      public boolean setLocalPort(int port);
      public int getLocalPort();
      public boolean receiveFile();
}
```

The `setMode` method specifies the algorithm used for reliable delivery where `mode` is 0 or 1 (to specify stop-and-wait and sliding window respectively). If the method is not called, the mode should default to stop-and-wait. Its companion, `getMode` simply returns an int indicating the mode of operation.

The `setModeParameter` method is used to indicate the size of the window in bytes for the sliding window mode. A call to this method when using stop-and-wait should have no effect. The default value should be 256 for the sliding window algorithm. **Hint:** your program will have to use this value and the MTU (max payload size) value to calculate the maximum number of outstanding frames you can send if using the Sliding Window algorithm. For instance, if the window size is 2400 and the MTU is 20 you could have up to 120 outstanding frames on the network.

The `setFilename` method is used to indicate the name of the file that should be sent (when used by the sender) or the name to give to a received file (when used by the receiver).

The `setTimeout` method specifies the timeout value in milliseconds. Its default value should be one second.

A sender uses `setReceiver` to specify IP address (or fully qualified name) of the receiver and the remote port number. Similarly, `setLocalPort` is used to indicate the local port number used by the host. The sender will send data to the specified IP and port number. The default local port number is 12987 and if an IP address is not specified then the local host is used.

The methods `sendFile` and `receiveFile` initiate file transmission and reception respectively. Methods returning a boolean should return true if the operation succeeded and false otherwise.

Operation requirements

RReceiveUDP

1. Should print an initial message indicating the local IP, the ARQ algorithm (indicating the value of n if appropriate) in use and the UDP port where the connection is expected.
2. Upon successful initial connection from a sender, a line should be printed indicating IP address and port used by the sender.
3. For each received message print its sequence number and number of data bytes
4. For each ACK sent to the sender, print a message indicating which sequence number is being acknowledged
5. Upon successful file reception, print a line indicating how many messages/bytes were received and how long it took.

RSendUDP

1. Should print an initial message indicating the local IP, the ARQ algorithm (indicating the value of n if appropriate) in use and the local source UDP port used by the sender.
2. Upon successful initial connection, a line should be printed indicating address and port used by the receiver.
3. For each sent message print the message sequence number and number of data bytes in the message
4. For each received ACK print a message indicating which sequence number is being acknowledged
5. If a timeout occurs, i.e. an ACK has been delayed or lost, and a message needs to be resent, print a message indicating this condition
6. Upon successful file transmission, print a line indicating how many bytes were sent and how long it took.

Fragmentation and in-order-delivery

In-order delivery is implicitly solved by the stop-and-wait algorithm (only one outstanding packet at all times) and explicitly by the sliding-window algorithm. You are responsible for developing an operating solution that will allow your programs to reconstruct the file even if fragments were received out-of-order.

Design

Your programs are essentially operating at the application layer (just like ftp, http and many other protocols) using an unreliable transport layer (UDP). Please plan your design before you start coding and define a packet format that will allow safe and consistent file transfers. Your header design must include information that will allow the receiver to detect when the file has been transferred. Furthermore, the receiver should save the received data as soon as the transfer is complete.

I encourage group discussions and sharing of the packet format used for the file transfers. Implementation should be individual. The advantage of sharing the packet format with other students is that you will have the ability of testing your program with others.

DO NOT make RSendUDP and RReceiveUDP part of any package.

Usage

Program usage will be illustrated by the following example. Assume Host A has IP address 192.168.1.23 and Host B has IP address 172.17.34.56 (note that none of these IP addresses are routable over the Internet). Host A wants to send a file named `important.txt` to Host B. Host A wants to use local port 23456 and Host B wants to use local port 32456 during the file transfer. Host B does not consider the file

so important and it wants to save the received file as `less_important.txt`. Both Host A and B have agreed to use the sliding-window protocol with a window size of 512 bytes and Host A will use a timeout value of 10 seconds (the path between A and B has a rather large delay).

Sample Code

The following code running on Host A should accomplish the task of reliably sending the file:

```
RSendUDP sender = new RSendUDP();
sender.setMode(1);
sender.setModeParameter(512);
sender.setTimeout(10000);
sender.setFilename("important.txt");
sender.setLocalPort(23456);
sender.setReceiver(new InetSocketAddress("172.17.34.56", 32456));
sender.sendFile();
```

The following code running on Host B should accomplish the task of receiving the file:

```
RReceiveUDP receiver = new RReceiveUDP();
receiver.setMode(1);
receiver.setModeParameter(512);
receiver.setFilename("less_important.txt");
receiver.setLocalPort(32456)
receiver.receiveFile();
```

Sample Output

Assume our solution uses 10 bytes of header for every message, the network cannot deliver messages larger than 200 bytes and the file we are transferring is 800 bytes long. The output of your program (on the sender side) when running the stop-and-wait algorithm could look like this:

```
Sending important.txt from 192.168.1.23:23456 to 172.17.34.56:32456 with 800 bytes
Using stop-and-wait
Message 1 sent with 190 bytes of actual data
Message 1 acknowledged
Message 2 sent with 190 bytes of actual data
Message 2 acknowledged
Message 3 sent with 190 bytes of actual data
Message 3 timed-out
Message 3 sent with 190 bytes of actual data
Message 3 acknowledged
Message 4 sent with 190 bytes of actual data
Message 4 acknowledged
Message 5 sent with 40 bytes of actual data
Message 5 acknowledged
Successfully transferred important.txt (800 bytes) in 1.2 seconds
```

A similar output should be seen on the receiver side.

Grading

Stop-and-wait and sliding window have to be implemented for full credit. Note that the value of the MTU cannot be changed programmatically when using `UDPSocket`. However, you can manipulate its value by editing the `unet.properties` file. The `getSendBufferSize` and `setSendBufferSize` provided by DatagramSocket (and consequently by UDPSocket) should be used when accessing MTU values (a call to setSendBufferSize in UDPSocket will throw an exception).

At the time your submission is graded, several values of MTU will be used (by modifying `unet.properties`). Please make sure you test your program thoroughly.

Report
A brief report should be typed and turned in by the due date at class time. The report does not need to be long and should only describe your approach (and challenges) in solving the problem and any methods or variables of relevance in your class. In particular, describe the packet format in detail and provide explanations for each field.
The report should also include a "feedback" section about this project.

Submission
The report and electronic submissions are due at class time on the due date

Submissions will be done via Harvey (harvey.utulsa.edu) and at a minimum, should contain your report, one RSendUDP.java file, one RReceiveUDP.java file and any other supporting classes you created.

Appendix: UDP Datagram transfers in Java (UNET Example Program)

The following two pieces of code illustrate how to use UDP facilities in Java. A sender program is used to send a "Hello World" message to a receiver over UDP. The UDPSocket class (in unet.jar) extends the DatagramSocket class provided by Java and gives you additional control over any message transfers as specified in the unet.properties file. Please note that these two programs will still work if all instances of UDPSocket are replaced by DatagramSocket.

Server

The following code creates a server that listens on the port number passed through the command line and simply outputs the data it receives.

```
import java.net.DatagramPacket;
import edu.utulsa.unet.UDPSocket; //import java.net.DatagramSocket;
import java.net.InetAddress;

public class udpreceiver {
   public static void main(String[] args)
   {
    int localPort=Integer.parseInt(args[0]);
    try
      {
       UDPSocket socket = new UDPSocket(localPort);
       while(true){
            byte [] buffer = new byte[80];
            DatagramPacket packet = new DatagramPacket(buffer,buffer.length);
            socket.receive(packet);
            InetAddress client = packet.getAddress();
            System.out.println(" Received'"+new String(buffer)+"' from "
+packet.getAddress().getHostAddress()+" with sender port "+packet.getPort());
                }}
           catch(Exception e){ e.printStackTrace(); }
        }
}
```

Client

The following code creates a client, capable of transmitting a simple message to the server. It uses three command line arguments: a local port, name or ip of node to send the message to and the remote port used by the receiver.

```
import java.net.DatagramPacket;
import edu.utulsa.unet.UDPSocket;
import java.net.DatagramSocket;
import java.net.InetAddress;
public class udpsender {
        public static void main(String[] args)
        {
          int localPort=Integer.parseInt(args[0]);
          String serverName=args[1];
          int serverPort=Integer.parseInt(args[2]);
                try {
                    byte [] buffer = ("Hello World- or rather Mauricio saying hello
through UDP").getBytes();
                 UDPSocket socket = new UDPSocket(localPort);
                 //DatagramSocket socket = new DatagramSocket(localPort);
                 socket.send(new DatagramPacket(buffer, buffer.length
                 InetAddress.getByName(serverName), serverPort));
                }
                catch(Exception e){ e.printStackTrace(); }
        }}
```

Compilation and Execution

We first compile these two Java files with the following commands:

```
D:\somedir>javac -classpath unet.jar;. udpsender.java
D:\somedir>javac -classpath unet.jar;. udpreceiver.java
```

Run the receiver first and then the sender (you will need two separate shells). In U*ix-based systems the separator for multiple classpath arguments is a : and not a ; like in Windows (so you'd type `javac -classpath unet.jar:.`). For this example we use the same host as the sender and receiver (using localhost):

```
D:\somedir>java -cp unet.jar;. udpreceiver 34567
 Received 'Hello World' from 127.0.0.1

D:\somedir>java -cp unet.jar;. udpsender 12345 localhost 34567
50      0 127.0.0.1:34567 --> Hello World :: 48 65 6C 6C 6F 20 57 6F 72 6C 64
```

The output generated by the client is part of the logging facility provided by edu.utulsa.unet.UDPSocket, and can be managed through unet.properties. This feature provides information regarding the status of packets within the UDPSocket object.