Programming Assignment 2

CS 544 – Computer Networks – Spring 2014
Building in Reliability

Due Date: June 10, 2014, at 11:59 PM
Assignments are to be done individually.
Acknowledgement to Raouf Boutaba for sharing materials

1. Assignment Objective

In the last assignment, you created a file-transfer protocol from client to server that was unreliable. The goal of this assignment is to learn and incorporate reliability into your application.

To this end, you will implement the **Go-Back-N (GBN)** protocol, which could be used to transfer a text file from one host to another across an unreliable network. The protocol should be able to handle network errors such as packet loss ~~and duplicate packets~~. For simplicity, your protocol will remain unidirectional, i.e., data will flow in one direction (from the client to the server) and the acknowledgements (ACKs) in the opposite direction. To implement this protocol, you will write two programs: a client and a server, with the specifications given below. You will test your implementation using an emulated network link (which will be provided to you) as shown in the diagram below:

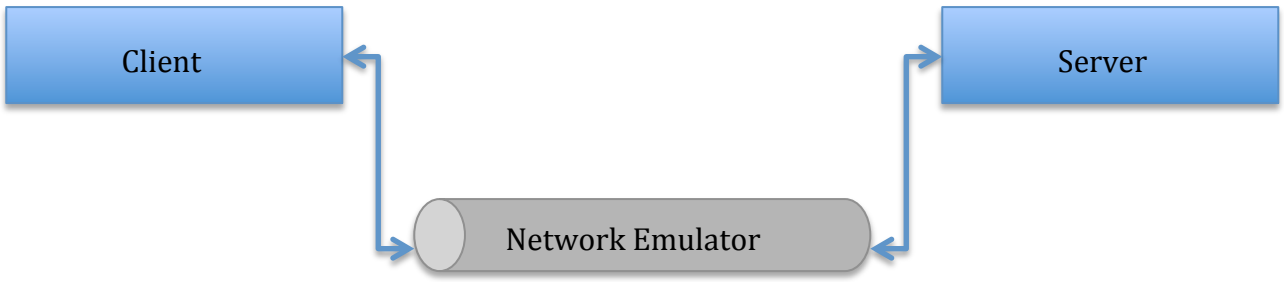

Figure 1

When the client sends packets to the server, it sends them to the network emulator instead of directly to the server. The network emulator then forwards the received packets to the server. However, it may randomly discard and/or delay received packets. The same scenario happens when the server sends ACKs to the client.

Note: The assignment description (data structure and program names) assumes an implementation in Java.

2. Packet Format

All packets exchanged between the client and the server should have the following structure (this class is provided):

```
public class packet {
     .........
     private int type;
     private int seqnum;
     private int length;
     private String data;
     .........
}
```

The `type` field indicates the type of the packet: it is set to 0 if it is an ACK, 1 if it is a data packet, 2 if it is an end-of-transmission (EOT) packet from server to client, and 3 if it is an EOT packet from client to server.

For data packets, `seqnum` is the modulo 8 sequence number of the packet. That is, sequence numbers have values in {0, 1, 2, 3, 4, 5, 6, 7}.

The sequence number of the first packet should be zero. For ACK packets, `seqnum` is one more than the sequence number of the packet being acknowledged.

The length field specifies the number of characters carried in the data field. It should be in the range of 0 to 30, taking the largest chunk of data per packet sent (i.e. take 30 if you can, less if you have reached the end of the file).

For ACK packets, length should be set to zero.

You may **not** modify this class (we will use our own packet class when testing).

3. Client Program (client)

You should implement a client program, named `client`. Its command line input includes the following:

```
<emulatorName: host address of the emulator>,

<sendToEmulator: UDP port number used by the emulator to receive data from
the client>,
```

```
<receiveFromEmulator: UDP port number used by the client to receive ACKs
from the emulator>,

<fileName: name of the file to be transferred>
```

in the given order.

Upon execution, the client program should be able to read data from the specified file and send it using the GBN protocol to the server via the network emulator. The window size should be set to N=7. After all contents of the file have been transmitted successfully to the server (and corresponding ACKs have been received), the client should send an EOT packet to the server. The EOT packet is in the same format (and it has a sequence number) as a regular data packet, except that its `type` field is set to 3 and its `length` is set to zero. The client can close its connection and exit only after it has received ACKs for all data packets it has sent and received an EOT from the server. To keep the project simple, you can assume that the end-of-transmission packet never gets lost in the network.

To ensure reliable transmission, your program should implement the GBN protocol as follows:

If the client has a packet to send, it first checks to see if the window is full, that is, whether there are *N* outstanding, unacknowledged packets. If the window is not full, the packet is sent and the appropriate variables are updated. A timer of 800 milliseconds is started if it was not done before. The client will use only a single timer that will be set for the oldest transmitted-but-not-yet-acknowledged packet. If the window is full, the client will try sending the packet later. When the client receives an acknowledgement packet with sequence number *n*, the ACK will be taken to be a cumulative acknowledgement, indicating that all packets with a sequence number up to and but not including *n* have been correctly received at the server. If a timeout occurs, the client resends all packets that have been previously sent but that have not yet been acknowledged. **If an ACK is received which is greater or equal to Sf**, but there are still additional transmitted-but-yet-to-be-acknowledged packets, the timer is restarted. If there are no outstanding packets, the timer is stopped.

3.1 Output

For both testing and grading purposes, your `client` program should be able to generate two log files, named as *seqnum.log* and *ack.log*. Whenever a packet is sent, its sequence number should be recorded in *seqnum.log*. The file *ack.log* should record the sequence numbers of all the ACK **(or EOT)** packets that the client receives during the entire period of transmission. The format for these two log files is one number per line. You must follow this format to avoid losing marks.

4. Server Program (server)

You should implement the server program, named `server`. Its command line input includes the following:

```
<emulatorName: hostname for the emulator>,

<receiveFromEmulator: UDP port number used by the server to receive data
from the emulator>,

<sendToEumulator-Port: UDP port number used by the emulator to receive ACKs
from the server>,

<fileName: name of the file into which the received data is written>
```

in the given order.

When receiving packets sent by the client via the network emulator, it should execute the following:

- check the sequence number of the packet;
- if the sequence number is the one that it is expecting, it should send an ACK packet back to the client with the sequence number one more than the sequence number of the received packet (indicating it is now awaiting the next sequence number);
- In all other cases, it should discard the received packet and resends an ACK packet for the most recently received in-order packet. After the server has received all data packets and an EOT from the client, it should send an EOT packet (type 2) then exit.

- ***4.1 Output*** The server program is also required to generate a log file, named as *arrival.log*. The file *arrival.log* should record the sequence numbers of all the data packets **(and EOT packets)** that the server receives during the entire period of transmission. The format for the log file is one number per line. You must follow the format to avoid losing marks.

5. Network Emulator (emulator**)** You will be given **the executable code** for the network (multithreaded) emulator. To run it, you need to supply the following command line parameters in the given order:

```
<receivePort: emulator's receiving UDP port number, used by both client and
server>,

<sendToClient-Port: client's receiving UDP port number>,

<sendToServer-Port: server's receiving UDP port number>,

<clientName: client's network address>,

<serverName: server's network address>,

<maxDelay: ~~maximum delay of the link in units of milliseconds~~>,
           Actually a seed for testing purposes now that we are not
           considering delay.

<dropProb: packet discard probability>,

<verbose-mode: int set to 1, the network emulator will output its internal
processing>
```

When the link emulator receives a packet from the client, it will discard it with the specified probability. ~~Otherwise, it stores the packet in its buffer, and later forwards the packet to the server with a random amount of delay (roughly the specified maximum delay).~~

Serialization and deserialization should be used to transmit and receive packets, respectively. Learn about the following commands:

```
ByteArrayOutputStream oSt = new ByteArrayOutputStream();
ObjectOutputStream ooSt = new ObjectOutputStream(oSt);
ooSt.writeObject(pkt);
ooSt.flush();
byte[] sendBuf = new byte[30];
sendBuf = oSt.toByteArray();
```

and the commands for deserialization. The following link seems like a good start (or read about it anywhere else you would like):

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

and you can see examples here:

http://stackoverflow.com/questions/17940423/send-object-over-udp-in-java

http://stackoverflow.com/questions/3997459/send-and-receive-serialize-object-on-udp-in- java

The emulator will respond to most incorrect inputs; of course, it may be ``breakable'' and you should not depend on it to catch errors in your transmitted packets.

An example of how to run the emulator is posted to Blackboard alongside this assignment as an attachment (this is just showing how you run the emulator).

6. Handing in the Assignment

6.1 Due Date

This assignment is due on June 10, 2014, at 11:59 PM. The penalty for late assignments was covered in the course syllabus and reviewed in Lecture 1.

6.2 Hand in Instructions

Submit all of your files in a single compressed file (either .zip or .tar) using Blackboard. You must hand in the following files **AND ONLY THE FOLLOWING FILES (not other folders, subfolders, text files, etc.)**:

- Source code files.
- Makefile: your code must compile and link cleanly by typing ``make''. This does not have to include the emulator and subemulator executables.

Here is an example execution:

```
java emulator 6000 6001 6002 tux64-11 tux64-13 0 0.0 1
java server tux64-12 6002 6000 output.txt
java client tux64-12 6000 6001 file.txt
```

where the emulator is running tux64-12, the server is running on tux64-13, and the client is running on tux64-11. I will also provide screenshots on Blackboard (under the "Programming Assignments" section).

Your implementation will be tested on tux, and so **it must run smoothly in this environment!** We will be unzipping your submitted file and running it with the commands as described above. **You will lose significant points if your code does not compile or execute, or does not conform to this execution style.**

6.3 Documentation

Since there is no external documentation required for this assignment, you are expected to have a reasonable amount of internal code documentation (to help the graders read your code).

7. Additional Advice

- Use port numbers at 1024 or above to avoid reserved port numbers

- Build and test your code incrementally. Start with no packet loss and no delay to start using localhost. Then, add packet loss ~~and delay~~. Then test using tux.

- Remember to start the emulator and server first before starting the client

~~- You will likely want to use multithreading for at least your client (since receiving is a blocking operation). To learn about multithreading, you can read this (or anything else you like):~~

~~http://tutorials.jenkov.com/java-multithreaded-servers/multithreaded-server.html~~

and of course, the following is always useful to learn more:

http://docs.oracle.com/javase/7/docs/api/

- Get started early. You have the better part of a month, but **do not** leave this to the last minute (or even the last few days).