Lab 4: Implementation of the Go Back N Protocol

1. Overview

In this laboratory programming assignment, **you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol**. You have to implement the **Go Back N Protocol**. This lab should be fun since your implementation will differ very little from what would be required in a real-world situation.

Since you probably don't have standalone machines (with an OS that you can modify), your code will have to execute in a simulated hardware/software environment. However, the programming interface provided to your routines, i.e., the code that would call your entities from above and from below is very close to what is done in an actual UNIX environment. Stopping/starting of timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

2. The Routines You will Write

The protocol you will write are for the sending entity (A) and the receiving entity (B). Only unidirectional transfer of data (from A to B) is required. Of course, the B side will have to send packets to A to acknowledge (positively or negatively) receipt of data. Your routines are to be

implemented in the form of the procedures described below. These procedures will be called by (and will call) procedures which emulate a network environment.

The unit of data passed **between the upper layers and your (transport layer) protocols** is a message, which is declared as an entity class in Message.java:

```
public class Message{
        private String data;
        ……..
        ……..
}
```

This declaration, and all other entity classes are stored in a package named "uofm.lab3.section.yourid.entity". In java Entity class is a class which holds some values and perform some operation on those values. Though String could hold any length data but in this assignment your sending entity will receive data in 20-byte chunks from layer5 (i.e. the application layer); your receiving entity should deliver 20-byte chunks of correctly received data to layer5 (i.e. application layer) at the receiving side.

The unit of data passed **between your routines and the network layer** is the *packet*, which is declared as:

```
public class Packet{
        private int seqnum;
        private int acknum;
        private int checksum;
        private String payload;
        ……..
        ……..

}
```

Your routines will fill in the payload field from the message data passed down from layer5 (aOutput(Message message) method will be called.). You will find those routine in the **StudentNetworkSimulator.java** class. The other packet fields will be used by your protocols to insure reliable delivery, as we have studied in class. The routines you will write are detailed below. As noted above, such procedures in real-life would be part of the operating system, and would be called by other procedures in the operating system. The relationship between some of these routines is shown in Figure 1.

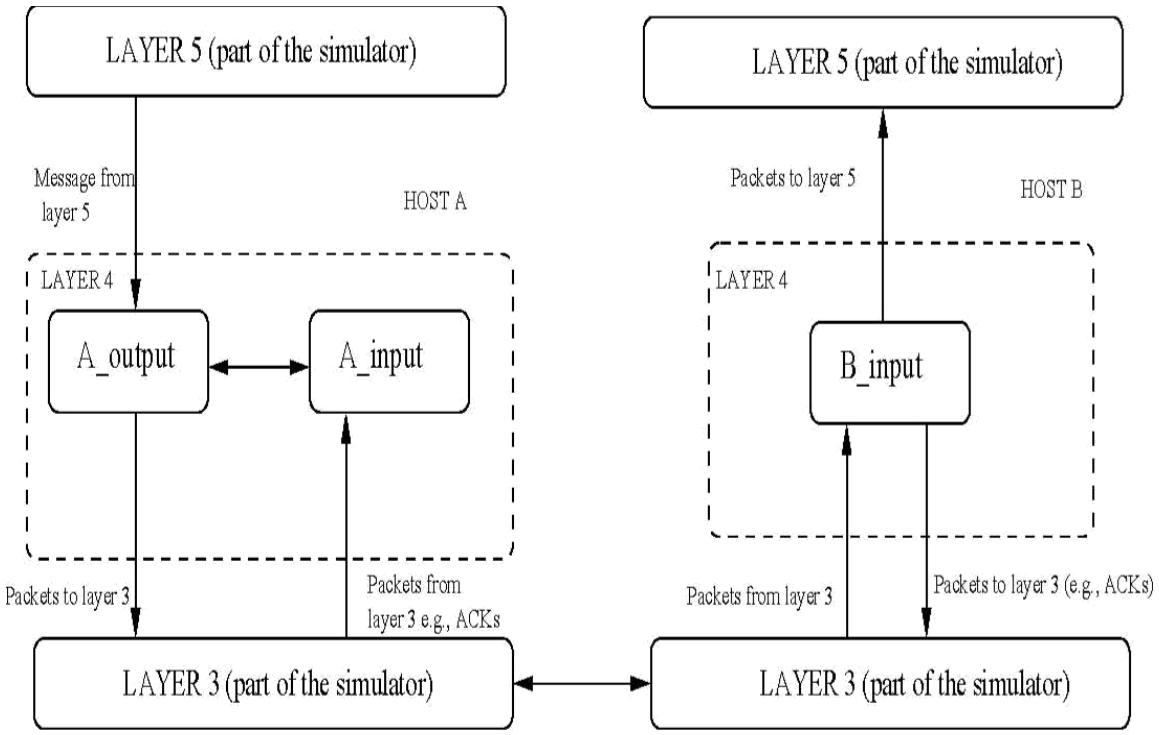

Figure1: Relationship between various routines

- **aOutput(Message message)** where `message` is an object of class **Message,** containing data to be sent to the B-side. This routine will be called whenever the upper layer at the sending side (A) has a message to send. It is the job of your protocol to insure that the data in such a message is delivered in-order, and correctly, to the receiving side upper layer.
- **aInput(Packet packet)** where `packet` is an object of class **`Packet`.** This method will be called whenever a packet sent from the B-side (i.e., as a result of a `tolayer3()` being done by a B-side procedure) arrives at the A-side. `packet` is the (possibly corrupted) packet sent from the B-side.
- **aTimerinterrupt():** This method will be called when A's timer expires (thus generating a timer interrupt). You'll probably want to use this routine to control the retransmission of packets. See `startTimer()` and `stopTimer()` method definition below for how the timer is started and stopped.

- **aInit():** This method will be called once, before any of your other A-side routines are called. It can be used to do any required initialization**.**

- **bInput(Packet packet):** Where `packet` is an object of class **`Packet.`** This method will be called whenever a packet sent from the A-side (i.e., as a result of a `tolayer3()` being done by a A-side procedure) arrives at the B-side. `packet` is the (possibly corrupted) packet sent from the A-side.

- **bInit():** This method will be called once, before any of your other B-side routines are called. It can be used to do any required initialization.

2. Software Interfaces

The procedures described above are the ones that you will write. I have written the following routines which can be called by your routines:

- **startTimer(calling_entity,increment),** where `calling_entity` is either 0 (for starting the A-side timer) or 1 (for starting the B side timer), and `increment` is a *float* value indicating the amount of time that will pass before the timer interrupts. A's timer should only be started (or stopped) by A-side routines, and similarly for the B-side timer. To give you an idea of the appropriate increment value to use: a packet sent into the network takes an average of 5 time units to arrive at the other side when there are no other messages in the medium.

- **stopTimer(calling_entity),** where `calling_entity` is either 0 (for stopping the A-side timer) or 1 (for stopping the B side timer).

- **tolayer3(calling_entity,packet),** where `calling_entity` is either 0 (for the A-side send) or 1 (for the B side send), and `packet` is an object of class `Packet.` Calling this routine will cause the packet to be sent into the network, destined for the other entity.

- **tolayer5(calling_entity,message),** where `calling_entity` is either 0 (for A-side delivery to layer 5) or 1 (for B-side delivery to layer 5), and `message` is a structure of type `msg.` With unidirectional data transfer, you would only be calling this with `calling_entity` equal to 1 (delivery to the B-side). Calling this routine will cause
  data to be passed up to layer 5.

  I have used some constant variables to minimizes complexity, those are:

```
public static final int MAXDATASIZE = 20; // as we are sending 20 bytes of data
```

```
// These constants are possible events
public static final int TIMERINTERRUPT = 0;
public static final int FROMLAYER5 = 1;
public static final int FROMLAYER3 = 2;

// These constants represent our sender and receiver
public static final int A = 0;
public static final int B = 1;
```

4. The Simulated Network Environment

A call to procedure `toLayer3()` sends packets into the medium (i.e., into the network layer). Your procedures `aInput()` and `bInput()` are called when a packet is to be delivered from the medium to your protocol layer.

The medium is capable of corrupting and losing packets. It will not reorder packets. When you compile your procedures and my procedures together and run the resulting program, you will be asked to specify values regarding the simulated network environment:

- **Number of messages to simulate.** My emulator (and your routines) will stop as soon as this number of messages have been passed down from layer 5, regardless of whether or not all of the messages have been correctly delivered. Thus, you need **not** worry about undelivered or unACK'ed messages still in your sender when the emulator stops. Note that if you set this value to 1, your program will terminate immediately, before the message is delivered to the other side. Thus, this value should always be greater than 1.

- **Loss.** You are asked to specify a packet loss probability. A value of 0.1 would mean that one in ten packets (on average) are lost.

- **Corruption.** You are asked to specify a packet loss probability. A value of 0.2 would mean that one in five packets (on average) are corrupted. Note that the contents of payload, sequence, ack, or checksum fields can be corrupted. Your checksum should thus include the data, sequence, and ack fields.

- **Tracing.** Setting a tracing value of 1 or 2 will print out useful information about what is going on inside the emulation (e.g., what's happening to packets and timers). A tracing value of 0 will turn this off. A tracing value greater than 2 will display all sorts of odd messages that are for my

own emulator-debugging purposes. A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that *real* implementors do not have underlying networks that provide such nice information about what is going to happen to their packets

- **Average time between messages from sender's layer5.** You can set this value to any non-zero, positive value. Note that the smaller the value you choose, the faster packets will be arriving to your sender.

5. The Go-Back-N Protocol

For the Go-Back-N protocol part of the lab, you are to write the methods, a`Output()`, `aInput()`, `aTimerinterrupt()`, `aInit()`, `bInput()`, and `bInit()` which together will implement a Go-Back-N unidirectional transfer of data from the A-side to the B-side, with a window size of 8. Your protocol should use both ACK and NACK messages. Consult the alternating-bit-protocol version of this lab above for information about how to obtain the network emulator.

We already implemented Alternating Bit protocol in Lab 3. So now you have to extend your code to implement this lab (Go-Back-N). However, some new considerations for your Go-Back-N code (which do not apply to the Alternating Bit protocol) are:

- **aOutput(Message message),** where `message` is an object of class Message, containing data to be sent to the B-side. Your aOutput() routine will now sometimes be called when there are outstanding, unacknowledged messages in the medium - implying that you will have to buffer multiple messages in your sender. Also, you'll also need buffering in your sender because of the nature of Go-Back-N: sometimes your sender will be called but it won't be able to send the new message because the new message falls outside of the window. Rather than have you worry about buffering an arbitrary number of messages, it will be OK for you to have some finite, maximum number of buffers available at your sender (say for 50 messages) and have your sender simply abort (give up and exit) should all 50 buffers be in use at one point (Note: using the values given below, this should never happen!) In the “real-world” of course, one would have to come up with a more elegant solution to the finite buffer problem!

- **aTimerinterrupt()** This method will be called when A's timer expires (thus generating a timer interrupt). Remember that you've only got one timer, and may have many outstanding,

unacknowledged packets in the medium, so you'll have to think a bit about how to use this single timer.

6. JAVA Files

The java files are organized in three packages named,
uofm.lab3.section.yourid.entity
uofm.lab3.section.yourid.logic
uofm.lab3.section.yourid.alternatebitprotocol
Note: Use your section and student id, for example, uofm.lab3.secb.7775147.logic.

 NetworkSimulator is an abstract class that is the bulk of the simulator. ***StudentNetworkSimulator*** is the **only class** that you have to modify.

 Packet, *Message*, *Event*, and *EventListImpl* are support classes. EventList is an interface. *Project* is the "driver" for the whole thing.

 You have to modify *StudentNetworkSimulator.java*. The other sources are there in case you are interested. *StudentNetworkSimulator.java* contains inline comments documenting the interfaces of the other classes that you will need.

7. Helpful Hints

- **Checksumming.** You can use whatever approach for checksumming you want. Remember that the sequence number and ack field can also be corrupted. We would suggest a TCP-like checksum, which consists of the sum of the (integer) sequence and ack field values, added to a character-by-character sum of the payload field of the packet (i.e., treat each character as if it were an 8 bit integer and just add them together).

- **Global state:** Note that any shared "state" among your routines needs to be in the form of global variables. Note also that any information that your procedures need to save from one invocation to the next must also be a global (or static) variable. For example, your routines will need to keep a copy of a packet for possible retransmission. It would probably be a good idea for such a data structure to be a global variable in your code. Note, however, that if one of your global variables is used by your sender side, that variable should **NOT** be accessed by the receiving side entity, since in real life, communicating entities connected only by a communication channel can not share global variables.

- **Time variable:** There is a float global variable called *time* that you can access from within your code to help you out with your diagnostics msgs.

- **Start simple:** Set the probabilities of loss and corruption to zero and test out your routines. Better yet, design and implement your procedures for the case of no loss and no corruption, and get them working first. Then handle the case of one of these probabilities being non-zero, and then finally both being non-zero.

- **Debugging.** We'd recommend that you set the tracing level to 2 and put LOTS of printf's in your code while your debugging your procedures.

- **Random numbers:** If you get an error message: *"It is likely that random number generation on your machine is different from what this emulator expects. Please take a*

look at the routine jimsrand() in the emulator code. Sorry." then you'll know you'll need to look at how random numbers are generated in the routine jimsrand(); see the comments in Section 8.

8. FAQ and Other Instructions

- **Frequently Asked Questions:** The authors of the textbook have pasted a FAH at: http://gaia.cs.umass.edu/kurose/transport/programming_assignment_QA.htm

- **Random number routine:** Random numbers are generated in the OSIRandom.java class.

9. Deliverables/Submissions

You should run the programs for a long time and plot graphs to show the effect of i) window size and ii) packet loss rate on the total throughput. You can save the outputs into a text file and extract data using a Java program and plot the graphs in Matlab.

You will need to submit the following:

- **Project** folder renamed as Lab3_7775148 where 7775148 is your id. The StudentNetworkSimulator.java file containing the routines you have written, appropriately commented. Note that your all package name **MUST** be changed according to your section and your student id to qualify for any credit.

- **A text file containing the output:** Insert System.out.println() statements at appropriate places in the code to obtain output in a format given below. Then you can copy and paste it in a text file. You can also BufferedWriter class to generate text file directly.

- Outlines the way packets have been transmitted and received in different lost settings. You should submit output for a run that was long enough so that at least 20 messages were successfully transferred from sender to receiver (i.e., the sender receives ACK for these messages) transfers in each case.

- **Different cases to be considered:** You will need to run your program under the following settings:

- Zero probability of loss and packet corruption.
- 0.1 probability of loss, zero probability of corruption.

- 0.1 probability of loss and 0.1 probability of corruption.
- Trace level set to 1 for all runs of the program.

- Put your project in a folder named "Section_YourId_Lab4" with a text file (readme.txt) contains your student information (Name, Id, Email) and any specific instruction you would like to notify the lab instructor. Also add the output files you have generated in different settings.

- Compress that folder in ".zip" format and submit it. Your submission would be like "SectionB_7775148_Lab4.zip" containing your project, output files and the readme.txt file.

- **You should work on Netbeans IDE since the given project is created by Netbeans 8.02.**

10. Sample Input Go Back N Protocol Output

-- * Network Simulator v1.0 * --
Enter number of messages to simulate (> 0): [10] 25
Enter packet loss probability (0.0 for no loss): [0.0] 0
Enter packet corruption probability (0.0 for no corruption): [0.0] 0
Enter average time between messages from sender's layer 5 (> 0.0): [1000] 1000
Enter window size (> 0): [8] 8
Enter retransmission timeout (>0.0) [15.0] 15
Enter trace level (>= 0): [0] 1
Enter random seed: [0] 749

A: Sending packet #1
B: Received packet #1
A: Sending packet #2
B: Received packet #2
A: Timeout
A: Retransmitting packet #1
A: Retransmitting packet #2
A: Sending packet #3
A: Packet #1 ACKed
A: Packet #2 ACKed
B: Discarding duplicate or out of order packet.
A: Packet #2 ACKed

. . . .

. . . .
B: Received packet #5
B: Received packet #6
A: Timeout
A: Retransmitting packet #5
A: Retransmitting packet #6
A: Packet #5 ACKed
B: Discarding duplicate or out of order packet.

A: Sending packet #7
B: Discarding duplicate or out of order packet.
A: Packet #6 ACKed
B: Received packet #7

Simulator terminated at time 282.3596911526841

After sending 25 packets.

- This is a sample output. You can generate your own format if you like.
- You do not have to print the payload or checksum or sequence number. Keep it simple.