CS78 - Programming Assignment 3: Implementing a Reliable Transport Protocol

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 will be provided with some skeleton code which has some functions that are to be implemented. The end result in the lab is to implement Go-Back-N that was described in the K&R text book. It is also possible to implement the Alternating-Bit Protocol with the provided skeleton software and this is recommended as an initial step to take as an approach to solving this assignment.

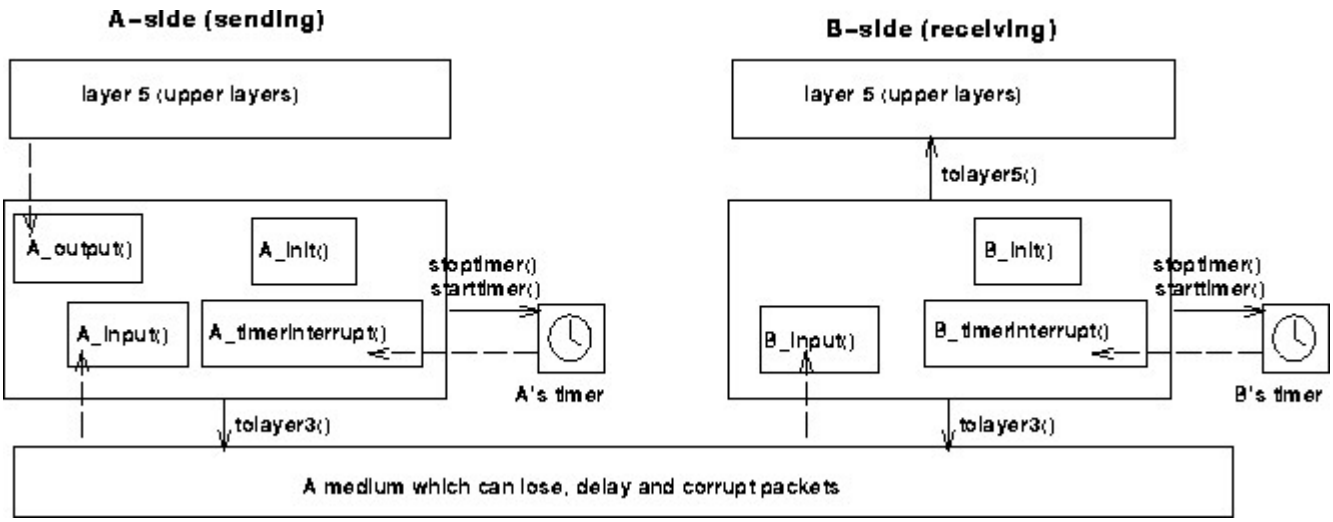
Please note that we will only require that Go-Back-N be implemented. However this document describes implementing the Alternating Bit Protocol first and then extending the code from here since other students doing this lab found this be a very useful step.

This lab will require you to code using C. Please do not be alarmed. As you are provided with skeleton code to modify you are not required to create code from the ground up. Depending on if you have coded with C previously you may need to consult C tutorials. However no advance aspects of C will be required to solve this lab. For the purposes of this assignment it will not find it all that removed from code you might have written in Java. Please consult the references at the end of this document and you should be fine.

Skeleton Code and the routines you will write.

The procedures 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 that have been written which emulate a network environment in other parts of the skeleton. You will only need to alter the functions that are clearly commented in the source code as being to be implemented by students. The rest of the source code should be unchanged. You make like to define some addition constant and variables beyond the function stubs you are provided. Equally you may want to implement some additional functions that are used by the empty function stubs in the source code. It is up to you how you decide to architect your solution.

The overall structure of the environment is shown below (structure of the emulated environment):



The unit of data passed between the upper layers and your protocols is a *message,* which is declared as:

```
struct msg {
  char data[20];
  };
```

This declaration, and all other data structure and emulator routines, as well as stub routines (i.e., those you are to complete) are in the skeleton file, **prog2.c,** described in more detail later. Your sending entity will thus receive data in 20-byte chunks from layer5; your receiving entity should deliver 20-byte chunks of correctly received data to layer5 at the receiving side.

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

```
struct pkt {
   int seqnum;
   int acknum;
   int checksum;
   char payload[20];
    };
```

Your routines will fill in the payload field from the message data passed down from layer5. The other packet fields will be used by your protocols to insure reliable delivery, as we've seen in class.

The routines you will write are detailed below.

- **A_output(message),** where `message` is a structure of type `msg,` 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 (not corrupted), to the receiving side upper layer.
- **A_input(packet),** where `packet` is a structure of type `pkt.` This routine 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.

- **A_timerinterrupt()** This routine 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()` below for how the timer is started and stopped.
- **A_init()** This routine will be called once, before any of your other A-side routines are called. It can be used to do any required initialization.
- **B_input(packet),**where `packet` is a structure of type `pkt.` This routine 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.
- **B_init()** This routine will be called once, before any of your other B-side routines are called. It can be used to do any required initialization.

Software Interfaces

The procedures described above are the ones that you will write. A number of routines have written for you to call from within the routines you implement, these are:

- **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 a structure of type `pkt.` 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.

The simulated network environment

A call to procedure `tolayer3()` sends packets into the medium (i.e., into the network layer). Your procedures `A_input()` and `B_input()` 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.** The 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 10% of packets are (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 be arriving to your sender.

An Alternating-Bit-Protocol Implementation

If you were implement an Alternating-Bit-Protocol you would need to write the procedures, `A_output()`,`A_input()`,`A_timerinterrupt()`,`A_init()`,`B_input()`, and `B_init()` which together will implement a stop-and-wait (i.e., the alternating bit protocol, which we referred to as rdt3.0 in the text) unidirectional transfer of data from the A-side to the B-side. **Your protocol should use both ACK and NACK messages.**

You should choose a very large value for the average time between messages from sender's layer5, so that your sender is never called while it still has an outstanding, unacknowledged message it is trying to send to the receiver. I'd suggest you choose a value of 1000. You should also perform a check in your sender to make sure that when `A_output()` is called, there is no message currently in transit. If there

is, you can simply ignore (drop) the data being passed to the `A_output()` routine.

You should put your procedures in a file called prog2.c. You will need the initial version of this file, containing the emulation routines that were written for you, and the stubs for your procedures. You can obtain this program from http://www.cs.dartmouth.edu/~cs78/prog2.c.

This lab can be completed on any machine supporting C. It makes no use of UNIX features. (You can simply copy the `prog2.c` file to whatever machine and OS you choose). There is no reason why it shouldn't be possible to do this lab under any OS such as Windows or using Mac. Regardless of the machine you are using it will need to have a ANSI C compiler installed.

We would encourage you to use the linux machines available in the cs department for this assignment. You can use these machines from your own computer by using SSH. All CS students typically will have accounts on these machines, if you do not please contact the system admins and they will provide you with an account. If you have any difficulty please contact me, Nic.

The source code provided, `prog2.c` will compile without modification. However until you implement the stub routines at are for you to implement it will do very little. You can compile it from a linux shell using the following commands:

```
> gcc prog2.c –o prog2
```

and then execute the compiled binary as so..

```
> ./prog2
```

Until you implement the stub routines you will be able to specific the network characteristics, and you will be able to see trace messages generated with packets being generated to be delivered but nothing else will occur. No packets will be delivered.

Please make sure you read the "helpful hints" for this lab at the bottom of this document.

Implementing Go-Back-N

You will write the procedures, `A_output()`,`A_input()`,`A_timerinterrupt()`,`A_init()`,`B_input()`, and `B_init()` 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.

We would **STRONGLY** recommend that you first implement first an Alternating Bit version of the lab and then extend your code to implement the harder lab (Go-Back-N). Others have found this not to be

wasted time at all. However, some new considerations for your Go-Back-N code (which do not apply to the Alternating Bit protocol) are:

- **A_output(message),** where `message` is a structure of type `msg,` containing data to be sent to the B-side.

  Your A_output() 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 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!

- **A_timerinterrupt()** This routine 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.

Submission of the Assignment

Please hand in as an email your source code, a short description of how you implemented your solution and a trace file that demonstrates your protocol in action. Please submit your short description as a hard copy as well. Comments will be written on this and returned to you.

Your trace file should contain output that shows 20 messages were successfully transfered from sender to receiver (i.e., the sender receives ACK for these messages) transfers, a loss probability of 0.2, and a corruption probability of 0.2, and a trace level of 2, and a mean time between arrivals of 10. You might want to annotate parts of your printout with a colored pen showing how your protocol correctly recovered from packet loss and corruption.

For **extra credit,** you can implement bidirectional transfer of messages. In this case, entities A and B operate as both a sender and receiver. You may also piggyback acknowledgments on data packets (or you can choose not to do so). To get my emulator to deliver messages from layer 5 to your B_output()

routine, you will need to change the declared value of BIDIRECTIONAL from 0 to 1.

Helpful Hints and the like

- **Checksumming.** We would like you to implement a form of checksum to allow you to determine if a packet was corrupted during it's transmission. You can use whatever approach 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).
- 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.
- 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.** The emulator generates packet loss and errors using a random number generator. Our past experience is that random number generators can vary widely from one machine to another. You may need to modify the random number generation code in the emulator we have supplied you. Our emulation routines have a test to see if the random number generator on your machine will work with our code. 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 that routine.

Q&A

When this lab has been given in other networking courses, students have posed various questions. If you are interested in looking at the questions that were received (and answers) by others doing a version of this assignment in other courses, check out http://gaia.cs.umass.edu/kurose/transport/programming_assignment_QA.htm

References for C programming

- http://www-ccs.ucsd.edu/c/index.html
- http://nrg.cs.ucl.ac.uk/mjh/3005/c-intro.pdf
- http://www.cs.princeton.edu/introcs/faq/c2java.html
- http://www.paulgriffiths.net/program/c/