CSE 489/589 Spring 2012
Programming Assignment 2
Implementing Reliable Transport Protocols

Due Time: 03/23/2012 @ 23:59:59
Optional for CSE489

1. Objective

Getting Started: Familiarize yourself with reliable transport protocols.
Implement: In a given simulator, develop 3 reliable data transport protocols - Alternating-Bit (AB, i.e., rdt3.0), Go-Back-N (GBN) and Selective-Repeat (SR).
Analyze: Understand how AB, GBN and SR work and compare their performance in the given simulator.

2. Getting Started

2.1 Alternating-Bit Protocol (rdt3.0)

Text book: Page. 224 – Page. 227

2.2 Go-Back-N Protocol

Text book: Page. 230 – Page. 235

2.3 Selective-Repeat Protocol

Text book: Page. 235 – Page. 242

3. Implement

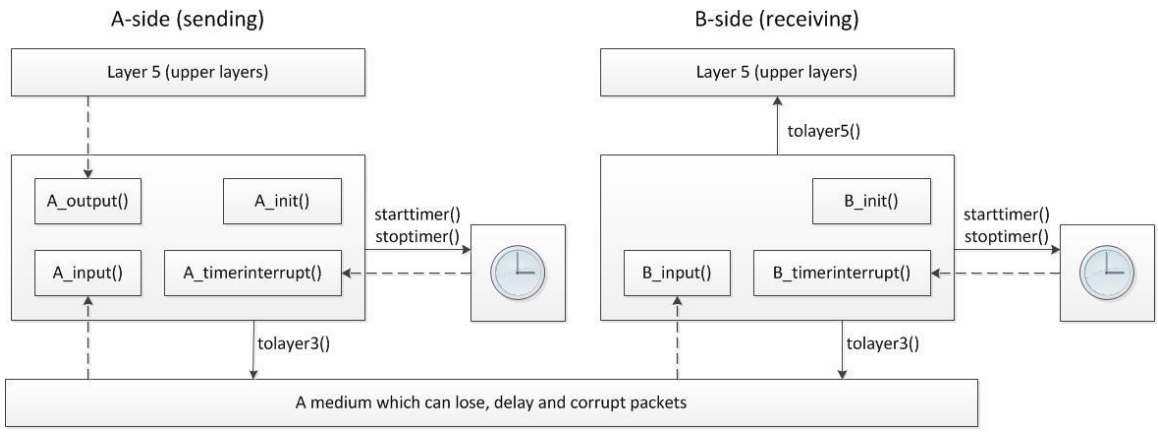
3.1 Overview

In this programming assignment, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. There are 3 versions of this assignment, the Alternating-Bit-Protocol version, the Go-Back-N version and Selective-Repeat version.

Since we 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. (Indeed, the software interfaces described in this programming assignment are much more realistic than the infinite loop senders and receivers that many texts describe). Stopping/starting of timers is also simulated, and timer interrupts will cause your timer handling routine to be activated.

3.2 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 which emulate a network environment. The overall structure of the environment is shown below:



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 file, prog2.c, described 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. 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.

- **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, 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.

Note: Those five routines are where you can implement protocols. For anywhere else, you are not supposed to modify any routines except for displaying your simulation results, which we will talk about later.

3.3 Software Interfaces

The procedures described above are the ones that you will write. We 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 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.

3.4 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 our 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 one in ten packets (on average) are lost.

- **Corruption**. You are asked to specify a packet corruption 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 our own emulator-debugging purposes. A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that, in reality, you would not have underlying networks that provide such nice information about what is going to happen to your 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.

3.5 The Alternating-Bit-Protocol Version

You are 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 is referred to as rdt3.0) unidirectional transfer of data from the A-side to the B-side. Your protocol should use only ACK 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. We 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 we have writen for you, and the stubs for your procedures. 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).

Before you start, make sure you read the "helpful hints" for this lab following the description of the Selective-Repeat version of this lab.

3.6 The Go-Back-N version

You are to 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 certain window size. Consult the alternating-bit-protocol version of this lab above for information about how to obtain the network emulator.

It is recommended that you first implement the easier lab (the Alternating-Bit version) and then extend your code to implement the harder lab (the Go-Back-N version). 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 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 500 messages) and have your sender simply abort (give up and exit) should all 500 buffers be in use at one point (Note: If the buffer size is not enough in your experiments, set it to a larger value) 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.

3.7 The Selective-Repeat version

You are to write the procedures, A_output(), A_input(), A_timerinterrupt(), A_init(), B_input(), and B_init() which together will implement a Selective-Repeat unidirectional transfer of data from the A-side to the B-side, with a certain window size.

It is recommended that you implement the former lab (Go-Back-N) before you extend your code to implement this lab (Selective-Repeat). Some new considerations for your Selective-Repeat code (which do not apply to Go-Back-N protocol) are:

- **B_input(packet)**, where packet is a structure of type pkt.

  You will have to buffer multiple messages in your receiver because of the nature of Selective-Repeat - the receiver should reply ACKs to all of the packets falling inside the receiving window. You can refer to the Go-Back-N version's A_output() to set your receiver's buffer.

- **A_timerinterrupt()** This routine will be called when A's timer expires (thus generating a timer interrupt). Even though the protocol uses many logical timers, remember that you've only got one hardware 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.

3.8 Helpful Hints and the like

- **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).

- 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 cannot 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 suplied 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.

4. Analyze

For analysis, we provide you a default setting to validate your protocols and then you can compare the 3 different protocols' performance with various loss probabilities, corruption probabilities and window sizes.

4.1 Validation of protocols

Run each of your protocols with a total number of 10 messages to be sent by entity A for the Alternating-Bit-Protocol version and 20 messages for the Go-Back-N and the Selective-Repeat versions. Set the loss probability to 0.2, the corruption probability to 0.2, the trace level to 2 and the mean time between messages arrivals (from A's layer5) to 1000 in the case of the Alternating-Bit-Protocol and 50 in the case of the other two protocols. For the Go-Back-N and the Selective-Repeat versions, set the window size to 10.

We recommend you to let your procedures print out a message whenever an event occurs at your sender of receiver (a message/packet arrival, or a timer interrupt) as well as any action taken in response. You might want to annotate some parts of your printout showing how your protocol correctly recovered from packet loss and corruption.

4.2 Performance comparison

In either of the following 2 experiments, run each of your protocols with a total number of 1000 messages to be sent by entity A , a mean time of 50 between messages arrivals (from A's layer5) and a corruption probability of 0.2.

- **Experiment 1:**
  With loss probabilities - {0.1, 0.2, 0.4, 0.6, 0.8}, compare the 3 protocols' throughputs at the application layer of receiver B. Use both 2 window sizes - {10, 50} for the Go-Back-N version and the Selective-Repeat Version.

- **Experiment 2:**
  With window sizes – {10, 50, 100, 200, 500} for GBN and SR, compare the 3 protocols' throughputs at the application layer of receiver B. Use both 2 loss probabilities – {0.2, 0.4} for all 3 protocols.

Please use the following format in you code to display the simulation result:
Protocol: [protocol_name]
[number_1] of packets sent from the Application Layer of Sender A
[number_2] of packets sent from the Transport Layer of Sender A
[number_3] packets received at the Transport layer
[number_4] of packets received at the Application layer

Total time: [time_] time units
Throughput = [number_4/time_] packets/time units

Note: You can assume there is no data delivery error between Layer 5 and Layer 4. However, handling down a message from Layer 5 to Layer 4 doesn't necessarily mean that the message is finally sent from Layer 4 to Layer 3. Remember that your simulator will stop when Layer 5 passes the specified number of messages down to Layer 4. To display simulation results, you have to modify main() which belongs to the simulator – add the results after line 221 in prog2.c. Keep in mind thus is the only part of the simulator code you are allowed to modify! You should follow the comments/guide of the stub code and should not modify the simulator except for displaying the results.

We recommend you to use a graph to show your results for each of the experiments in 4.2 and then write down your observations. What variations did you expect for throughput variations by changing those parameters and why? Do you agree with your measurements; if not then why?

5. Submission

5.1 What to Submit

Your submission should contain a tar file – A2.tar containing:

- All source files.
- A README describing how to compile and execute your source files for 3 protocols.
- Your printout for section 4.1 in a file named Analysis-A2-1.txt/pdf
- Your analysis (graphs and observations) for sections 4.2 in a file named Analysis-A2-2.txt/pdf, which should contain at least 2 graphs.

5.2 How to submit

Use the submission command, submit_cse489 or submit_cse589, to submit the tar file.

Appendix: Q&A

We provide solutions for the problems you may meet in this programming assignment:

1. The gcc compiler complains about the use of exit() in your code.

Change exit() to exit(0) if this is a problem.

2. The compiler complains about the use of the time variable in your code.

Some compilers will do this. You can change the name of my emulator's time variable to simtime or something like that.

3. My timer doesn't work. Sometimes it times out immediately after I set it (without waiting), other times, it does not time out at the right time. What's up?

My timer code is OK (hundreds of students have used it). The most common timer problem I've seen is that students call my timer routine and pass it an integer time value (wrong), instead of a float (as specified).

4. You say that we can access you time variable for diagnostics, but it seems that accessing it in managing our timer interrupt list would also be useful. Can we use time for this purpose?

Yes.

5. The jimsrand() function is not returning the right values on the my machine

To make jimsrand() work on the edlab Alphas, the variable mmm should be set to RAND_MAX (it may be necessary to include stdlib.h to get this constant).

6. Why is there a timer on the B side?

This is there in case you want to implement bi-directional data transfer, in which case you would want a timer for the B->A data path.

7. How concerned does our code need to be with synchronizing the sequence numbers between A and B sides? Does our B side code assume that Connection Establishment (three-way handshake) has already taken place, establishing the first packet sequence number ? In other words can we just assume that the first packet should always have a certain sequence number? Can we pick that number arbitrarily?

You can assume that the three way handshake has already taken place. You can hard-code an initial sequence number into your sender and receiver.

8. When I submitted my assignment I could not get a proper output because the program core dumped..... I could not figure out why I was getting a segmentation fault so ....

Offhand I'm not sure whether this applies to your code, but it seems most of the problems with seg. faults on this lab stemmed from programs that printed out char *'s without ensuring those pointed to null- terminated strings. (For example, the messages -- packet payloads -- supplied by the network

emulator were not null-terminated) This is a classic difficulty that trips up many programmers who've recently moved to C from a safer language.