Reliable Transport Protocol

CS536

Due: October 21, 2018, 11:59pm

1 Objective

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 three versions of this assignment, the bidirectional alternative-bit version, unidirection Go-Back-N version and unidirection Selective Repeat version. You need to implement all of these.

2 Implementation

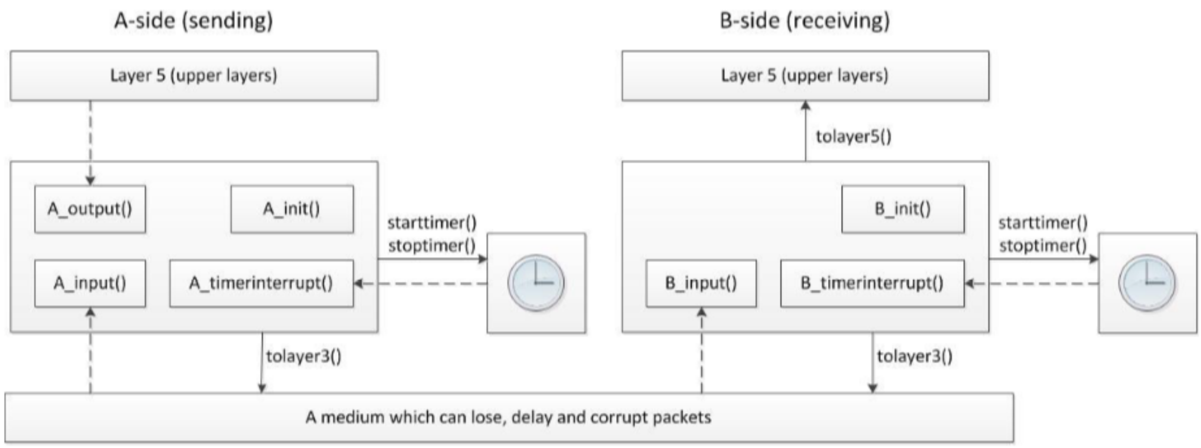
2.1 Programming Environment:

Your code should be able to compile and run correctly on the XINU machines (xinu1.cs.purdue.edu, xinu2.cs.purdue.edu,...) To SSH into those machine, in terminal type
ssh {your purdue username}@xinu{1-10}.cs.purdue.edu

For this assignment since we dont 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 are also simulated, and timer interrupts will cause your timer handling routine to be activated.

2.2 Routines you have to 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 provided to you which emulate a network environment. A diagram of the procedure 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, prog.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;
};
```

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 ensure reliable delivery. The routines you will write are detailed below. 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 ensure 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 called 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 As timer expires (thus generating a timer interrupt). Youll 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_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 (B) has a message to send in alternative bit protocol. It is the job of your protocol to ensure that the data in such a message is delivered in-order, and correctly, to the receiving side upper layer.

- **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 called by a A-side procedure) arrives at the B-side. packet is the (possibly corrupted) packet sent from the A-side.

- **B_timerinterrupt()**This routine will be called when Bs timer expires in alternative bit protocol (thus generating a timer interrupt). Youll use this routine to control the retransmission of packets. See starttimer() and stoptimer() below for how the timer is started and stopped.

- **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.

2.3 Software Interfaces

The procedures described above are the ones that you will write. We have written the following routines which will 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. As 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.

2.4 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**: Our 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 unACKed 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. whats 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 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 senders 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.

2.5 The Alternating-Bit-Protocol version

You are to write the procedures, A_output(), A_input(), A_timerinterrupt(), A_init(),B_output(), B_input(), B_timerinterrupt(), B_init(), which together will implement a stop-and-wait (i.e., the alternating bit protocol) **bidirectional** transfer of data from the A-side to the B-side and B-side to A-side. Your protocol can use both ACK and NACK messages (you can only use ACK messages as well). **Make sure your code allows unidirectional and bidirectional mode to be turned on and off, so we could perform analysis later on to compare with other protocols. You should start with single direction, then work on the bidrectional version of this protocol. Only Alternating-Bit-Protocol requires bidirectional communication.**

You should choose a very large value for the average time between messages from senders layer5, so that your sender is never called while it still has an outstanding, unacknowledged message it is trying to send to the receiver. Id 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_arq.c. The file, containing the emulation routines and the stubs for your procedures prog2.c is available on blackboard.

You should hand in a sample output along with your source code. For your sample output, your procedures must print out a message whenever an event occurs at your sender or receiver i.e on a message/packet arrival or a timer interrupt as well as any action taken in response. You should also print the content of the messages and payload of packets. You have to hand in output for a run up to the point (approximately) when 10 messages have been ACKed correctly at the receiver, a loss probability of 0.1, and a corruption probability of 0.3

and a trace level of 2. You might want to annotate your printout with a colored pen showing how your protocol correctly recovered from packet loss and corruption.

2.6 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 window size of 8. Your protocol can use both ACK and NACK messages. Consult the alternating- bit-protocol version of this assignment above for information about how to obtain the network emulator. We would STRONGLY recommend that you first implement the easier version (Alternating Bit) and then extend your code to implement the harder version (Go-Back-N). Believe us, it will save your time! 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, youll also need buffering in your sender because of the nature of Go-Back-N: sometimes your sender will be called but it wont 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 As timer expires (thus generating a timer interrupt). Remember that youve only got one timer, and may have many outstanding, unacknowledged packets in the medium, so youll have to think a bit about how to use this single timer.

You should put your procedures in a file called **prog2_gbn.c**.

You have to hand in 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, 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.

2.7 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 versions A_output() to set your receivers 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.

Again you have to hand in 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, 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.

3 Analysis

Now that we have implemented the Alternative bits, Go-Back-N and Selective Repeats protocols, we want to know which one is performing better. You are going to perform the two experiments using the unidirection protocol. 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 As layer5) and a corruption probability of 0.2.

3.1 Experiment 1:

With loss probabilities - {0.1, 0.2, 0.4, 0.6, 0.8}, compare the 3 protocols throughput 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.

3.2 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 doesnt 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 220 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 3.1, 3.2 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?

3.3 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).
- 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) ariable. 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 first and test your routines. Better yet, start out by designing and implementing 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.
- 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.

4 Submission and grading

1. You will be submitting your assignment on blackboard. Your submission directory should contain:
   (a) All source files i.e. prog2_arq.c, prog2_gbn.c and prog2_sr.c
   (b) Annotated output for as output_arq.pdf, output_gbn.pdf, and output_sr.pdf.
   (c) Another PDF lab2_experiment.pdf that contains the documentation of your experiments and your answers to the questions.

(d) A README file that contains instruction to run your code to demonstrate all the features required

2. Note:

   (a) Please document any reasonable assumptions you make or information in this file, e.g., if any parts of your assignment are incomplete.

   (b) Questions about the assignment should be posted on Piazza.