CS 1652 – Data Communication and Computer Networks – Project#2[1]

Due: Friday July 12th @ 11:59pm

Late submission: Sunday July 14th @11:59pm with 10% penalty per late day

OVERVIEW

Purpose: To implement a reliable transport protocol.

Goal 1: To implement the stop-and-wait protocol.

Goal 2: To implement the Go-Back-N protocol.

More specific details follow below.

OVERVIEW

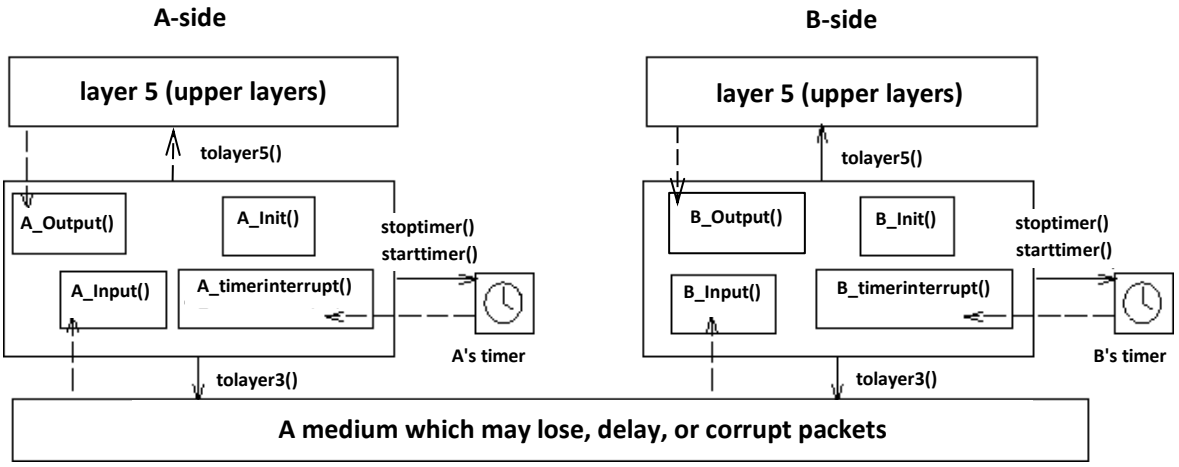
In this project, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. There are two parts of this project, the Alternating-Bit-Protocol (Stop-and-wait) and the Go-Back-N. This project should be fun since your implementation will 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.

THE ROUTINES YOU WILL WRITE

The procedures you will write are for two communicating entities, A and B, exchanging data in both directions. Of course, both sides will have to send control packets to each other 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 are provided for you and which emulate a network environment. The overall structure of the environment is shown in the below figure.

[1] Adapted from Computer Networking: a Top-Down Approach, 7th Edition.



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

A_timerinterrupt() 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() 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 A-side. This routine will be called whenever the upper layer at the sending side (B) 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.

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_timerinterrupt() will be called when B'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.

B_init() 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. The following routines are provided for you and 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. 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 the provided 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 provided 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 the provided emulator's 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.

THE ALTERNATING-BIT-PROTOCOL

You are to write the procedures, A_output(), A_input(), A_timerinterrupt(), A_init(), B_Output(), B_input(), B_timerinterrupt(), 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() (or B_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() (B_Output()) routine.

THE GO-BACK-N PROTOCOL

You are to write the procedures, A_output(), A_input(), A_timerinterrupt(), A_init(), B_Output(), B_input(), B_timerinterrupt(), and B_init() which together will implement a Go-Back-N bidirectional transfer of data between 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 project above for information about how to obtain the network emulator.

We would STRONGLY recommend that you first implement the easier part (Alternating Bit) and then extend your code to implement the harder part (Go-Back-N). Believe me - it will not be time wasted! However, some new considerations for your Go-Back-N code (which do not apply to the Alternating Bit protocol) are:

- **A_output(message) and B_Output(message)**, where message is a structure of type msg, containing data to be sent to the other 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 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() and B_timerinterrupt().** These routines will be called when either timer expires (thus generating a timer interrupt). Remember that you've only got one timer at each side, 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.

HELPFUL HINTS

1. **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).
2. 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 one side, that variable should NOT be accessed by the other side, since in real life, communicating entities connected only by a communication channel cannot share global variables.

3. There is a float global variable called *time* that you can access from within your code to help you out with your diagnostics msgs.
4. **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.
5. **Debugging**. We'd recommend that you set the tracing level to 2 and put LOTS of printf's in your code while you're debugging your procedures.
6. **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. The supplied 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.

7. You may also piggyback acknowledgments on data packets (or you can choose not to do so).

FREQUENTLY ASKED QUESTIONS

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

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

2. *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?*

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

3. *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.

4. *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.

5. *You instruct us to use both ACK and NAK messages. How would we signify a NAK event given the pkt struct you defined?*

A hack would be to use negative numbers for NAKs. You may also include an explicit ACK/NAK bit in the packet definition.

6. *In a bidirectional implementation, how would the sender indicate that a packet includes only an ACK and not an ACK plus data (when there's no data available on which to piggyback the ACK)?*

There would be no need to use the sequence number (or you could set it to a default value like -1) if there is no data in the packet.

7. *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 segmentation faults on this project 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, such as Java.

SUBMISSION REQUIREMENTS

You should put your procedures in the files **project2_stop_wait.c and project2_gbn.c**. You will need the initial version of these files, containing the emulation routines we have provided for you, and the stubs for your procedures. You can obtain this program from the project page on CourseWeb.

This project 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).

You should upload to the project page on **Gradescope** a **code listing, a design document, and sample output**. For your sample output, your procedures should print out a message whenever an event occurs at your sender or receiver (a message/packet arrival, or a timer interrupt) as well as any action taken in response.

- For the stop-and-wait part, you should hand in output for a run up to the point (approximately) when 10 messages have been ACK'ed correctly at the receiver, a loss probability of 0.1, and a corruption probability of 0.3, and a trace level of 2. You should annotate your sample output with colors showing how your protocol correctly recovered from packet loss and corruption.
- For the Go-Back-N part, you should hand in output for a run that was long enough so that at least 20 messages were successfully transferred from each side to the other side (i.e., the sender receives ACK for these messages), 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 should annotate your

sample output with colors showing how your protocol correctly recovered from packet loss and corruption.

GRADE BREAKDOWN

Item	Number of Points
Alternating-Bit Protocol	
*_Input()	10
*_Output()	10
*_init	5
*_timerinterrupt()	15
Design document and sample output	10
Go-Back-N Protocol	
*_Input()	10
*_Output()	10
*_init	5
*_timerinterrupt()	15
Design document and sample output	10