CS422 - Spring 2020
Programming Assignment 2
Reliable Transport Protocol
Due Date: March 31, 2020 11:59 pm   Total points: 100

I. OBJECTIVES

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 Alternating-Bit-Protocol version, the Go-Back-N version, and the Selective Repeat. You need to implement all of these.

II. IMPLEMENTATION

A. *Programming environment*

You must work individually on this assignment. You will write C code that compiles and operates correctly on the XINU machines (xinu01.cs.purdue.edu, xinu02.cs.purdue.edu, etc.) found in HAAS 257.

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

B. *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 provided to you which emulate a network environment. The overall structure of the environment is shown in the following figure:
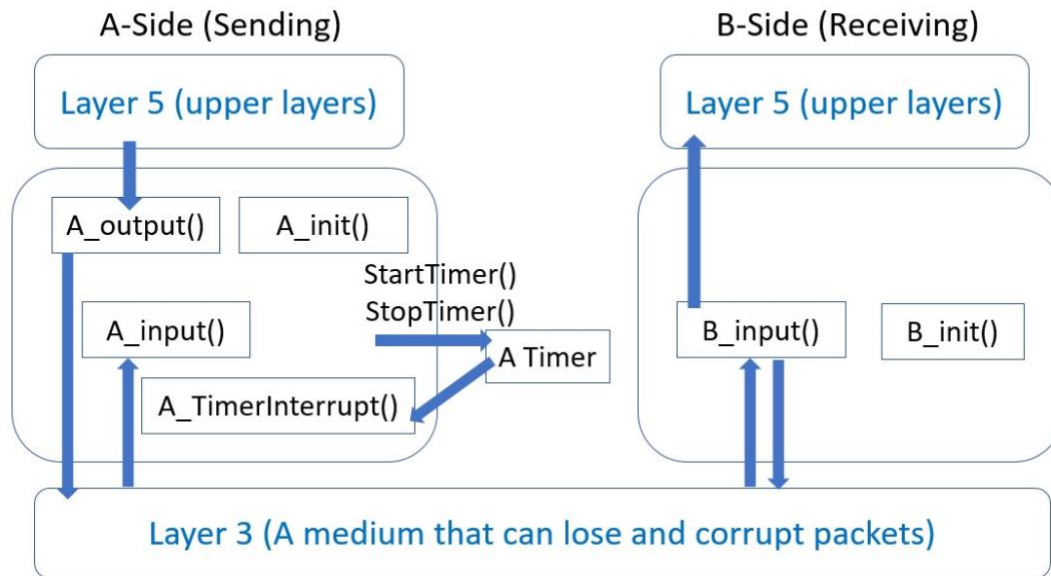
The unit of data passed between the upper layers and your protocols is a message, which is declared as:

struct msg {
    char data[MESSAGE_LEN];   // MESSAGE_LEN=20
};

This declaration and all other data structure that you may choose to use are in the file, **prog.h.** 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[MESSAGE_LEN];
};

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(adata)**, where adata is some additional data you can use to further distinguish timers on the same calling entity. This routine will be called when A's timer expires (thus generating a timer interrupt). You'll 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 called 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.

C. 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, adata),** where calling entity is either 0 (for starting the A-side timer) or 1 (for starting the B-side timer), increment is a float value indicating the amount of time that will pass before the timer interrupts, and adata is some additional data you can use to further distinguish

timers on the same calling entity. 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. **Note** that if you only need one timer on a calling entity, you may set adata to NULL.

- **stoptimer(calling entity, adata),** where calling entity is either 0 (for stopping the A-side timer) or 1 (for stopping the B-side timer) and adata is some additional data you can use to further distinguish timers on the same calling entity.
- **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.

D. 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 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 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 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.

E. The Alternating-Bit-Protocol version (40 points)

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) unidirectional transfer of data from the A-side to the B-side. Your protocol can use both ACK and NACK messages (you can only use ACK messages as well).

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 **tasks_alt.c.**

You should hand in a sample output along with your source code as output_alt.pdf. 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 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 might want to highlight how your protocol correctly recovered from packet loss and corruption.

F. Go-Back-N version. (30 points)

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, 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() This routine will be called when A's timer expires (thus generating a timer interrupt). Although there may be many outstanding, unacknowledged packets in the medium, you only need one timer for this protocol. So you'll have to think a bit about how to use this single timer.

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

You should submit output as output_gbn.pdf 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 highlight how your protocol correctly recovered from packet loss and corruption.

G. Selective Repeat version. (30 points)

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 window size of 8. Your protocol can use both ACK and NACK messages. However, compared with the Go-Back-N version, Selective Repeat avoids unnecessary retransmissions by retransmitting only those packets that were lost or corrupted. Unlike GBN, out-of-order packets are buffered by the receiver until any missing packets with lower sequence numbers are received. On the sender side, some packets in the window will have already received an ACK.

The implementation of this SR version may include (but is not limited to) the following changes. (a) Besides the window on the sender side, your receiver also needs a window to receive and record out-of-order packets. (b) Once the missing packets are received, a batch of in-order packets will be delivered to the upper layer (layer 5). (c) A-side should now have multiple timers since only a single packet will be transmitted on timeout. Please read **Session 3.4** (especially Figure 3.24 and Figure 3.25) in the textbook and make sure you understand how the protocol works. To buffer messages falling outside of the window, you can use a buffer of the same size (50) as in the GBN version. You can assume that the sequence number of the first packet sent out is 0.

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

You should submit output as output_sr.pdf 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 highlight how your protocol correctly recovered from packet loss and corruption.

H. 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) 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 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 nonzero and then finally both being non-zero.
- We put data structure and function declaration in **prog2.h**; the emulator routines in **prog2.c**; and stub routines (i.e. those you are to complete) in **tasks.c**. You may change prog2.h and prog2.c while implementing only for debugging purpose but should be aware that these two files will be replaced (including the parameters in them) after submission for grading.
- Compile your tasks.c together with emulator prog2.c by using the following line
    gcc prog2.c tasks.c -o PA2
- **START EARLY.** Since the three versions are not equally difficult, please start early. You may find yourself spending more time on debugging the GBN or SR version than you expect.

III. SUBMISSION AND GRADING

A. *What to Submit*

You will be submitting your assignment on blackboard. Your submission file (named as "YourPurdueID_lab2.zip") should contain:
- All source files i.e. **tasks_alt.c, tasks_gbn.c and tasks_sr.c**
- Sample output files i.e. **output_alt.pdf**, **output_gbn.pdf** and **output_sr.pdf**.

Note:
- Please document any reasonable assumptions you make or information in a **ReadMe** file, e.g. if any parts of your assignment are incomplete.
- If your programs need any special flags or commands for compiling, please submit a **Makefile** with your code.

Questions about the assignment should be posted on Piazza. In most of the cases, you might find your question already answered by some existing posts. There will be a post summarizing frequently asked questions. Please make sure you check it. If you need help understanding the concepts or the requirement of the project, you can contact Siyuan Cao at cao208@purdue.edu.