CS 555: Computer Communications and Networking (Spring 2019)

PA-2: Reliable Data Transfer (RDT)
Due date: 4/7, 11:59pm

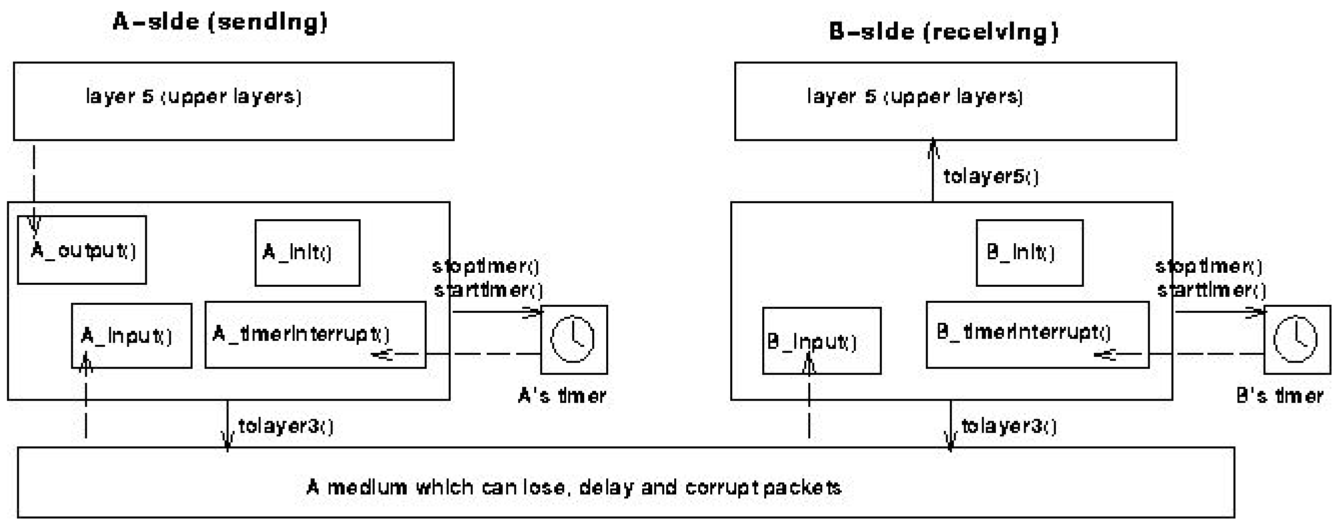
Project description

In this assignment, you will be writing the sending and receiving transport-layer code for implementing a simple reliable data transfer protocol. There are two parts in this assignment, the Alternating-Bit-Protocol version and the Go-Back-N version. As you know GBN forms the basis of TCP, you are actually creating a simulation of TCP in this programming assignment.

You have been provided with the programming interface with different routines/functions. These functions are actually similar to what TCP implements in a typical UNIX/Linux 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 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 we have written which emulate a network environment. The overall structure of the environment is shown in the figure 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.

Network emulator API

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.

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.** My 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 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.

Part 1: Alternating-Bit-Protocol

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 we referred to as rdt3.0 in the text) unidirectional transfer of data from the A-side to the B-side. **Your protocol should only use ACK (no NACK messages as in rdt 3.0).**

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 we have written for you, and the stubs for your procedures. You can obtain this program from ~~http://gaia.cs.umass.edu/kurose/transport/prog2.c~~
http://www.phpathak.com/CS555/prog2.c

Part 2: Go-Back-N (GBN)

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. As discussed in class and in textbook (Figs. 3.20 and 3.21), your protocol should only use ACK messages (no NACKs).

We would STRONGLY recommend that you first implement the easier lab (Alternating Bit) and then extend your code to implement the harder lab (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),** 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). 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.

Helpful hints and some design related FAQs

- **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 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: I can't compile the code on Windows developer studio
  A: Please use Linux and gcc.

- Q: "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?"
  A: Our 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).

- Q: Why is there a timer on the B side?
  A: you can ignore that, this is in case you have a lot of extra time after completing this project :-) and would want to implement bidirectional data transfer (B->A). No extra credit though.

- Q: "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?"
  A: 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.

Policies and submission

Programming language

You can implement your client in C as you have been provided with network emulator code which is written in C.

Working with a partner

You are required to do this project with your project partner of PA1. You can choose to do it alone but cannot change the partner from PA1 without my permission.

Note on plagiarism

In this class, it is absolutely mandatory that you adhere to GMU and Computer Science Department honor code rules. This also means (1) do not copy code from online resources and (2) your implementation should be purely based on your own thought process and conceived design, and not inspired by any other students or online resources, (3) you can copy your code or design from other students in the class.

We reserve the right to check your assignment with other existing assignments (from other students or online resources) for cheating using code matching programs. Any violation of honor code will be reported to the GMU honor committee, with a failing grade (F) in this course.

*** Do not put your code online on Github or other public repositories ***
Violation of this policy would be considered an honor code violation.

Grading, submission and late policy

- Standard late policy applies - Late penalty for this lab will be 15% for each day. Submissions that are late by 2 days or more will not be accepted
- This lab accounts for **9%** of your final grade
- You will submit your solution via Blackboard

What to submit? [Read this before you start working on the project]

- Submit the following
    1. Two C code files - “abp.c” and “gbn.c” for alternating bit protocol and go back n protocol, respectively. You can create two copies of the prog2.c file provided to you and name them “abp.c” and “gbn.c” referring to alternating bit protocol and go back n protocol, respectively. Submit both “abp.c” and “gbn.c” files.
    2. Two diff files: Run the following commands to generate diff files and submit the two diff files
       ```
       %> diff abp.c prog2.c > abp.diff
       %> diff bgn.c prog2.c > bgn.diff
       ```
    3. A README.txt which explains how to compile your code. Your code will be compiled using the following commands. If your code requires any specific instructions to compile, it should be included in the README.txt. Your assignment will be tested and

graded on a standard Linux machine. So, if your code requires any specific libraries, make sure to include that in the compilation instructions.

```
%> gcc abp.c -o abp
%> gcc bgn.c -o bgn
```

4. A design document which describes how you have implemented both the protocols. The document should discuss how you approached the problem, should point to specific functions and what they do, and any limitation your code has.
5. An OUTPUT.txt file that shows the output you get when running with the following parameters. Note that your code (built on prog2.c) does not take any arguments, so it is run simply using

   ```
   %> ./abp
   %> ./bgn
   ```

   When you run this code, you will be asked to provide input for (1) number of messages to simulate, (2) loss probability, (3) corruption probability, (4) average time between messages from the sender's layer5 and (5) trace level. Check the init() function of prog2.c for more information about the input.
   - Include the output of your abp and bgn programs for a run that was long enough so that at least 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.
6. A PARTNERS.txt file mentioning the name and GMU IDs of two students worked on the project as a team. Both team members should submit these four pieces separately on Blackboard before deadline.