Project 2: Implementing Reliable Data Transfer Protocol
(CS3516 – A18)

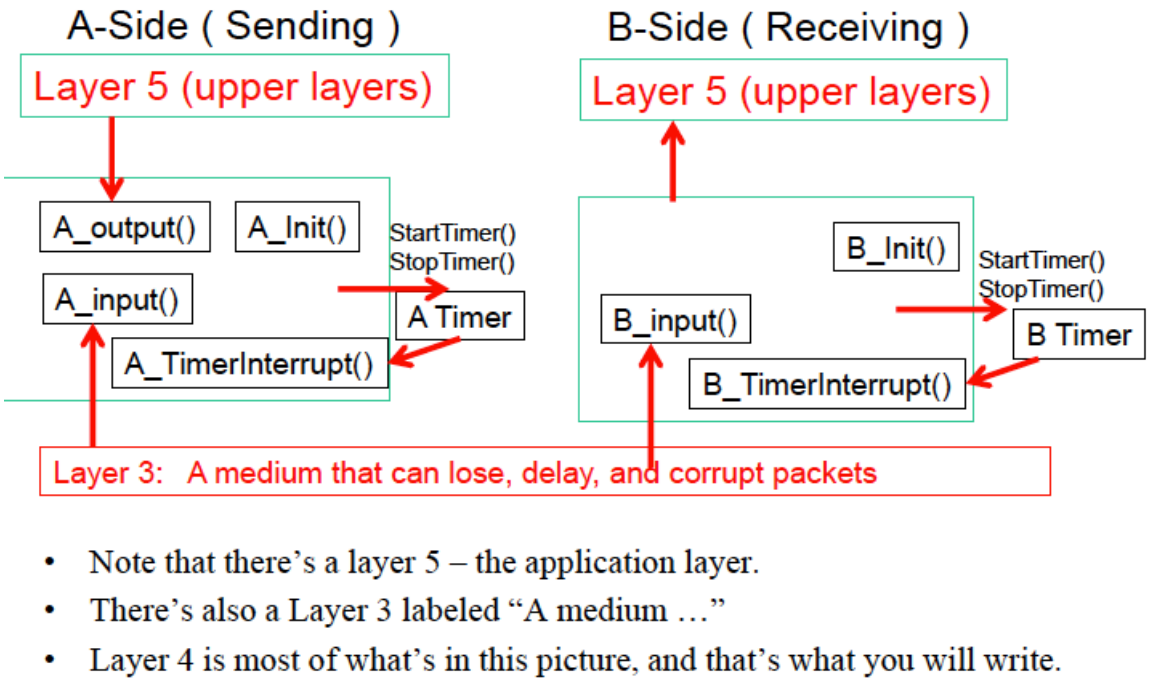
Due: Sep 28, 2018 F, 11:59 PM

Total Points
100 (One Hundred)

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, i.e., **the Alternating-Bit-Protocol (ABP).** 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 Linux 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.

Figure 1 below presents an overview of the elements you will be coding in this project. Our purpose of this project we only have three layers in our network stack.



- Note that there's a layer 5 – the application layer.
- There's also a Layer 3 labeled "A medium …"
- Layer 4 is most of what's in this picture, and that's what you will write.

Figure 1: Project Overview

The procedures you will write are for the sending entity (A) and the receiving entity (B). The equivalent reverse (bidirectional) traffic originates on the B-side and is received on the A-side. 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 in Figure 1. These procedures will be called by (and will call) procedures already written that emulate a network environment.

2. Communication Between Layers – Data Structures

Figure 2 below shows the data structure used for communicating between the layers

The unit of data passed between the upper layers (5) and your protocols in layer 4, is a *message*, which is declared as:

```
struct msg {
     char data[20];
};
```

It is expected that you will be able to handle this data in your Layer 4 routines. The content of the msg is pretty much irrelevant to you – your job is simply to get this 20-character string to the other side. Note – there may be embedded nulls in this data.

The definition of these structures, along with other goodies, is given in project2.h.

The unit of data passed between your protocols in Layer 4, and the network layer(3) 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.

Figure 2: Data Structure for Communicating between Layers

3. Routines you will write

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.

`void A_output(struct msg 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.

`void A_input(struct pkt 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. The arriving packet may be corrupted w.r.t. packet sent from the B-side.

`void 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.

`void 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.

`void B_input(struct pkt 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.
`void 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.

`void B_timerinterrupt();`
This routine 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.

`void B_output(struct msg message);`
Similar to **`A_Output()`** Required only when implement bi-directional messaging. **Ignore for this project.**

4. Available routines, to be used by your routines

The procedures described above are the ones that you will write. The following routines, which can be called by your routines:

`void startTimer(int AorB, double TimeIncrement);`
Where **`calling_entity`** is either **`AEntity`** (for starting the A-side timer) or **`BEntity`** (for starting the B side timer), and T**`imeIncrement`** is a double 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.

`void stopTimer( int AorB );`
Where **`calling_entity`** is either **`AEntity`** (for stopping the A-side timer) or **`BEntity`** (for stopping the B side timer).

`double getClockTime();`
Returns the current simulation time, which is very useful in printouts of traces since the simulator is already printing out such times.

`int getTimerStatus( int AorB );`
Returns **`TRUE`** if the requested timer is running, or **`FALSE`** if the timer is not running.

`void toLayer3( int AorB, struct pkt packet );`
Where **`calling_entity`** is either **`AEntity`** (for the A-side send) or **`BEntity`** (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.

`void toLayer5( int AorB, struct msg datasent);`
Where **`calling`**_entity is either **`AEntity`** (for A-side delivery to layer 5) or BEntity (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 **`BEntity`** (delivery to the B-side). Calling this routine will cause data to be passed up to layer 5.

5. Input Arguments

The medium is capable of *corrupting, losing, and reordering* packets. When you compile your

procedures and simulation procedures together, and run the resulting program, you will be asked to specify values regarding the simulated network environment. Figure 3 illustrates a sample input and below that you will find a description for the various input elements.

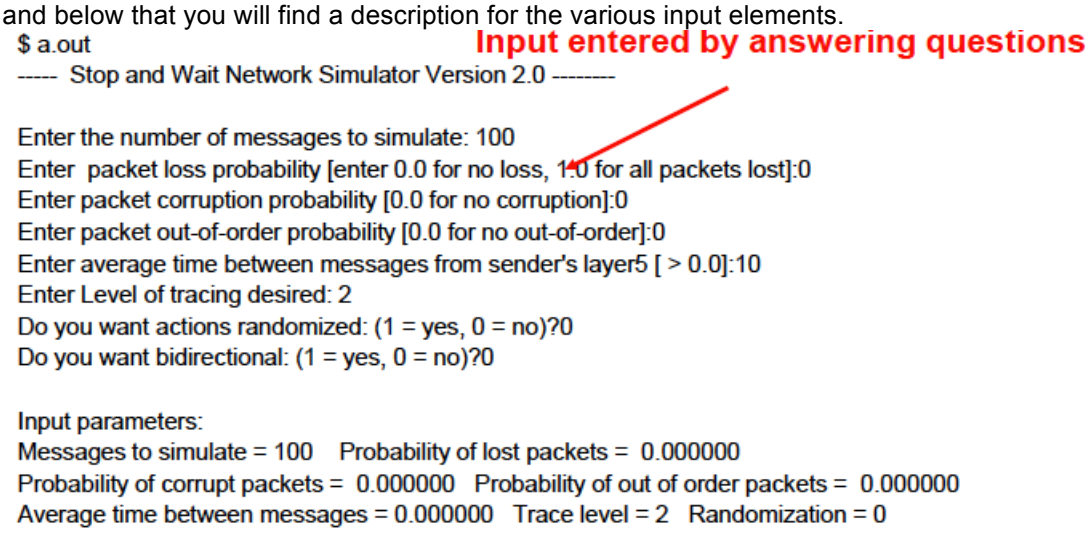


Figure 3: Input Arguments

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 and not delivered to the destination.

Corruption:
You are asked to specify a packet loss probability. A value of 0.2 would mean that two in ten 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.

Out Of Order:
You are asked to specify an out-of-order probability. A value of 0.2 would mean that two in ten packets (on average) are reordered.

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 of 5 will display all sorts of odd messages that are for emulator-debugging purposes. A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that real implementers do not have underlying networks that provide such nice information about what is going to happen to their packets! You will certainly find tracing your own code is helpful. When the time comes to show off your code, you must have a way of turning off all your debugging messages. (We will be running with tracing = 1, so you can set your messages to be displayed only for a higher tracing level – like 3 or 4.

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.

Randomization:
The simulation works by using a random number generator to determine if packets will or will not be modified in some fashion. Setting 0 here (no randomization) means that you will get the same result for each of your runs. This can be extremely valuable for debugging. However, for real testing, you must run with randomization = 1 to see what problems you can shake out. When you demonstrate your code, I expect to see randomization enabled.

Direction:
The possibilities are Unidirectional = 0, Bidirectional = 1 (Which should not be needed).

6. Environment

There are three files that you will use to implement your solution:
`project2.c` – contains the simulation code for the network layer and for the application layer 5.
`student2.c` – contains the stub of the numerous routines you are to write.
`project2.h` – various definitions and data structures that are included in both of the source code modules.
You can download the project 2 package (including these environment code) from here
https://users.wpi.edu/~yli15/courses/CS3516Fall18A/projects/Proj2/project_2_A18.zip

Note that this simulation runs on both Windows and LINUX. You should modify the location in Project2.h that specifies the OS. However your project will be evaluated on the ccc/rambo machines using Linux.

Compile the sources using **`gcc -g project2.c student2.c`**

You may want to divide the file student2.c into three components – my description implies you create three source files – but the details are up to you.

1. **`student2A.c`** contains the functions having well known names for the A Entity. The interfaces are here, the state information is here, but the routines that do all the work are kept in the common routine.
2. **`student2B.c`** contains the functions having well known names for the B Entity. The interfaces are here, the state information is here, but the routines that do all the work are kept in the common routine.
3. **`student_common.c`** contains all the code common to both A and B. Since A and B are really identical, the methods are here. But don't keep any state information here.

7. Actual Project

Now that you have an idea of the simulation environment and what you need to modify, we can talk about what you need to implement.

Alternating bit Protocol (RDT 3.0)
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 in Section 3.4.1) 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() is called, there is no

message currently in transit. If there is, you will need to buffer the message until the previous transaction is completed.

8. Helpful Hints

Checksumming: You can use whatever approach for checksumming you want. Remember that the sequence number and ack field can also be corrupted. A simple addition of all the bytes in the packet will NOT work – this is because I have diabolically defined corruption to be the swapping of bytes from two locations in the packet – a simple addition will give the same sum even with the packet corrupted.

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 double global variable called CurrentSimTime that you can access from within your code to help you out with your diagnostics messages. It represents the time as understood by the simulation.

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 visible with debug level = 2. The output needs to be clean when we look at it. We will be running with debug_level = 1.

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\n");
- Is different from what this emulator expects. Please take\n");
- A look at the routine GetRandomNumber() in the emulator code. Sorry.

Then you will need to sort out the routine.

9. Evaluation Criteria

Alternating Bit Protocol:

- Is there a clean output, free from messy debugging messages?
- Does the project work with corruption on?
- Does the project work with lost packets?
- Does the project work with packet out-of-order?
- Does the project work with randomization?

Others:

- Is the code commented? Is it free of numerical constants sprinkled in the code? Is it indented correctly?

- Is there a REAL makefile both phases of the project?
- Is there is a sufficient README file for both phases of the project?

10. Submission

As a part of the submission you need to provide the following:

For Alternating-bit-Protocol (ABP), create a zip file **"your-wpi-login_project2_ABP.zip"**

1. All the code for the alternating-bit protocol (any .c/.c++ and .h files), including any code that was pre-provided as part of this project
2. A ReadMe (txt) file that spells out exactly how to compile and run the code
3. A PDF document with the output trace as described in the project description

Submit your document electronically via Canvas (https://canvas.wpi.edu) by 11:59pm on the day the assignment is due. Make sure you choose "Project 2" under project drop-down in Canvas before uploading the zip file.

11. Grading

Grade breakup (%) assuming all documents are present:

1. Functioning Alternating Bit-Protocol code = 75%
2. Alternating Bit-Protocol output trace (as explained in the project description) = 15%
3. README = 5%
4. Makefile = 5%

The grade will be a ZERO if:

1. If the code does not compile or gives a run-time error
2. If README with detailed instructions on compiling the running the code is not present