CNT4704: Analysis of Computer Communication Networks (Fall 2015)

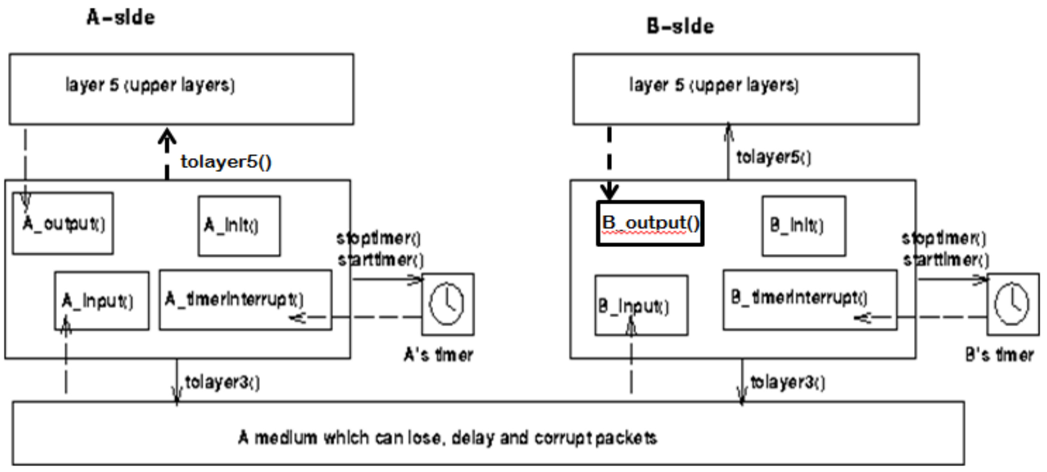
Project 2: Reliable Data Transfer Protocol

(Assigned Oct. 5th; due: Oct. 18th midnight)

In this second programming assignment, you will be writing the sending and receiving transport-level code for implementing the Alternating Bit Protocol (rdt3.0, showed on Page 4-6 on lecture slides 'Chapter3-part2.ppt') for ***bi-directional communication***. You should enjoy this assignment, as your implementation will differ very little from what would be required in a real-world situation.

Since we do not have standalone machines with an Operating System that you can modify, your code will have to execute in a simulated environment provided by the textbook (a simulator in C language). However, the programming interface provided to your routines (i.e., the calls to/from your code from/to the layer above - passing/receiving data to/from the application layer; and the calls to/from your code from/to the layer below - passing/receiving data to/from the layer below) is very close to what is done in an actual UNIX environment. Starting/stopping of timer are also simulated, and timer interrupts will cause your timer handling routine to be activated.

Your routines are to be implemented in the form of the procedures as described below. These procedures will be called by (and will also make calls to) procedures that the textbook authors have written which emulate a network environment which can cause packet error and packet loss. The overall structure of the environment is shown in the following:



The simulator has provided the functions of **tolayer5(), tolayer3(), stoptimer(), starttimer().** Your task is to complete the coding of the following routines:

Node A's functions: **A_output(), A_init(), A_input(), A_timerinterrupt()**

Node B's functions: **B_output(), B_init(), B_input(), B_timerinterrupt()**

The unit of data passed between the upper layer and your protocol is a **message**, which is declared in the simulator as:

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

Your sending node (either node A or node B) will thus receive data in 20-byte chunks from layer 5 (upper layer); your receiving node should deliver 20-byte chunks of correctly received data to layer 5 at the receiving side.

The unit of data passed between your routines and the network layer is a **packet**, which is declared in the simulator as:

```
struct pkt {
  int seqnum;   int acknum;   int checksum;   char payload[20];
   };
```

Your routines A_output() and B_output() will be called (activated) by simulator periodically to pass data **message** to the other node in the form of **packet**. These two functions should fill in the payload field of a packet from the message data passed down from layer 5. The other packet fields (3 fields) will be used by your protocol to insure reliable delivery, as we've seen in class.

The routines you will write are detailed below (we only use node A sends data to node B as example). 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 (node 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. This routine should return a value of 1 if this data packet was accepted for transmission, and -1 if this time's data transmission is rejected. Returning value of -1 will inform layer 5 that your protocol is busy trying to send previous data. In this case, layer 5 will reattempt to send this data at a later point in time.

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.

Software Interfaces

The procedures described above (and corresponding B-side routines) are the ones that you will write. The provided simulator has written the following routines which can (and should) 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 unit time 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

The procedure tolayer3() sends a packet 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 (can be configured). However, it will not reorder packets. When you compile the complete simulator with your codes in it and run it, you will be asked to specify the following values regarding the simulated network environment:

Number of messages to simulate. The simulator (and your routines) will stop as soon as a predefined number of messages have been passed down from layer 5 to your protocol, regardless of whether or not all of the messages have been correctly delivered.

Loss. You are asked to specify a packet loss probability. A value of 0.1 would mean that one in ten packets (on average) is lost.

Corruption. You are asked to specify a packet corruption probability. A value of 0.2 would mean that one in five packets (on average) have their bits corrupted. Note that the contents of the payload, sequence, ack, or checksum fields can all be corrupted. You must implement a checksum mechanism that covers all fields in a packet struct (see hint for checksum).

Tracing. Setting a tracing value of 1, 2 or 3 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 emulator-debugging purposes (but that could also help you debug your code).

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 before your sender is ready to send a new packet. I recommend using value of 20 for this variable to give a sender enough time to obtain feedback from the receiver.

You should put your procedures inside the file called simulator.c attached to this project, which contains the simulation engine. You will need the initial version of this file (simulator.c), containing the emulation routines, and the stubs for your procedures.

This assignment can be completed on any machine supporting C, whether Unix or Windows.

1. The Basic Assignment: Ideal Network Condition --- No loss, no packet corruption

You are to write code to implement a stop-and-wait protocol for a **bidirectional** transfer of data between node A and node B. When setting simulation parameters, we set it to have **10 messages to simulate, 0 loss probability, 0 corruption probability**, **trace level 2**.

Because this simulation assumes perfect channel, you are allowed to implement **rdt1.0** introduced in class to finish the data transfer.

2. The Advanced Assignment: Realistic Network Condition with loss and packet corruption

You are to write code to implement a stop-and-wait protocol (i.e., the Alternating Bit protocol, which we referred to as **rdt3.0** on the textbook and class notes), for a **bidirectional** transfer of data between node A and node B. When setting simulation parameters, we set it to have **10 messages to simulate, 0.1 loss probability, 0.2 corruption probability**, **trace level 2**.

What to turn in

Submit a zip file contains: (1). A project report. (2) the source code simulator1.c and simulator2.c for the two assignments; (3) explain how a TA can compile and run your code, and where your program can run (on Eustis machine, or a Windows computer).

To show me that you did successfully accomplished the assignment, in your project report: (1) describe in your report your overall program design with explanations for the design choices you might have made; (2) Screenshot image showing the execution progress of your simulator for the two assignments. If one screenshot is not enough, then paste several screenshot images that can be concatenated together to show the whole simulation progress. (3). Annotate on the screenshot images when each data message has been successfully received by the receiver side.

Helpful Hints

Checksumming. You can use whatever approach for checksumming you want. Remember that the sequence number and ack field can also be corrupted. I would suggest a simple summation-based 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 cannot share global variables.

Debugging. I'd recommend that you put LOTS OF PRINTF() statements in your code in some critical places while you debugging your procedures.

Packet Type Determination. Because both Node A and Node B will send data packets to each other, and also Ack packets to each other, we need a way to let a node tell whether a received packet is a data packet or an Ack packet. A simple way is to assign a special value (such as -10) to the "seqnum" field for an Ack packet since an Ack packet does not need to use sequence number. Another way is to fill special value in the payload[20] field to make it clear that the payload is not a normal data payload.

Of course, you can combine a data packet and the Acknowledgement together into one packet. It is more complex, but will save the total number of transmitted packets. This is the way how TCP implement bidirectional communication.