Department of CSE, BUET
CSE322: Computer Networks Sessional
Level 3, Term 2, July 2018 Term

Offline Assignment 4

Implementing Reliable Transport Protocols

1. Overview

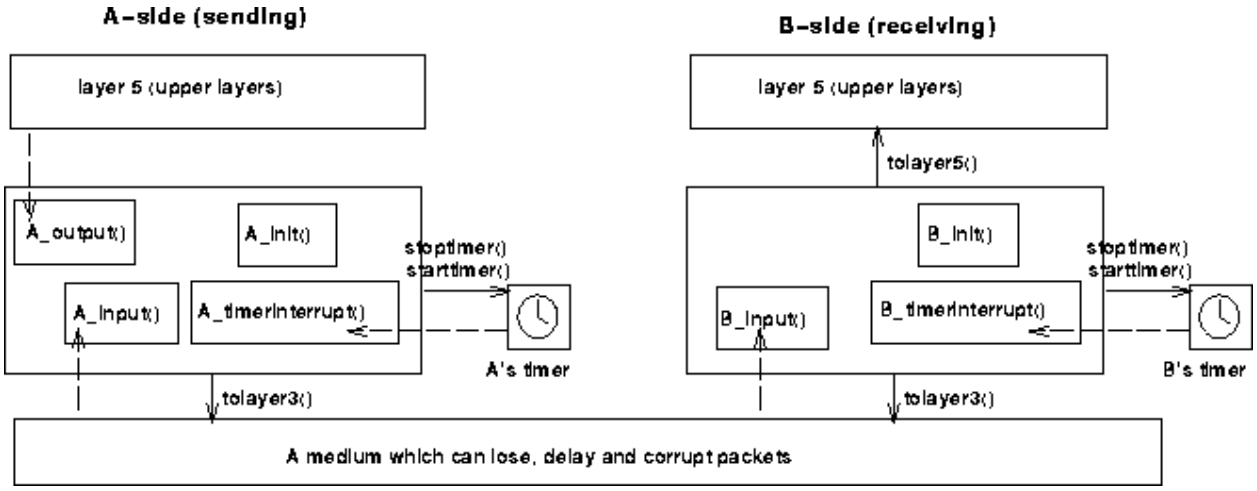
In this laboratory programming assignment, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. There will be two versions of this assignment that will take you progressively from easier to difficult level. The two versions are as follows: (i) alternating-bit protocol and (ii) Go-Back-N. This lab 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. (Indeed, the software interfaces described in this programming assignment are much more realistic that 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.

You are given a skeletal code and a network emulator as well in a file called **rdt.c**. You have to add your code in this given file. In the last lab you have worked with NS-2. From studying the emulator's code you will also learn how to write such simulators using discrete event simulation; though this is not an objective of this lab.

2. 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 have been written in the network emulator. The overall structure of the environment is shown in the figure below (structure of the emulated environment).



The unit of data passed between the upper layers and your protocols is a *message,* which is declared as:

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

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 the 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.

3. Software Interfaces

The procedures described above are the ones that you will write. However, the following procedures have already been written 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.

4. 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 given 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 emulator (and your routines) will stop after this number of messages have been transmitted from entity (A) to entity (B).
- **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 arriving to your sender.

5. Implementation of 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 use both ACK and NACK messages. Since there is not type field in the packet, NACK has to be implemented as by sending ACK again for the last correctly received packet.

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 **rdt_abp.c**.

You also need to provide a .**doc** file containing sample output from your code. The name of the file will be **output_abp.doc**. For sample output, your procedures might 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. You might want 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 need to annotate your **.doc** file with 'highlighting' to show how your protocol correctly recovered from packet loss and corruption.

6. Implementation of Sliding Window Protocol: Go-Back-N

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. You have to use finite sequence numbers for your packets, from 0-7. Determine the size of the window accordingly. Your protocol should use both ACK and NACK messages. The receiver will send NACK messages on the following two events: (i) the received packet is not expected, and (ii) the received packet has checksum error. The sender on receiving a NACK message (i.e., a duplicate ACK), will retransmit the current window.

The expected actions in response to all possible events for both the sender and the receiver are given below.

Table 1: Go-Back-N Sender Events and Actions

Event	Action
Data received from above	**Case 1:** the next available sequence number is within the sender's window - Packetize data and send. - Start a separate timer for the packet sent. **Case 2:** the sender's window is already full **Case 2.1:** the buffer is not full - Save data in the buffer for letter transmission. **Case 2.2:** the buffer is full - Refuse data.
Packet received from below	**Case 1:** the packet is not corrupted **Case 1.1**: 'acknum' in the incoming packet is within the window - Mark all the packets up to the 'acknum' as accepted. - Stop all the timers corresponding to the accepted packets. - Slide window to the next sequence number after 'acknum'. - If there are untransmitted packets (in the buffer) with sequence numbers that now fall within the window, transmit those packet. Start separate timers for each of the packets sent. **Case 1.2**: packet is a duplicate ack - Retransmit all the frames in the current window. - Restart timers for all the packets sent. **Case 2:** the packet is corrupted, i.e., checksum error - Ignore/drop packet.
Timeout	- Retransmit all the frames in the current window - Restart timers for all the packets sent

Table 2: Go-Back-N Receiver Events and Actions

Event	Action
Packet received from below	**Case 1:** the packet is not corrupted **Case 1.1**: packet is expected - Decapsulate the packet and send the extracted message to network layer. - Send ACK for the packet received. **Case 1.2**: packet is not expected - Send NACK, i.e., a duplicate ACK for the last correctly received packet **Case 2:** the packet is corrupted, i.e., checksum error - Send NACK, i.e., a duplicate ACK for the last correctly received packet

We would **STRONGLY** recommend 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 drop/refuse the message should all 50 buffers be in use at one point. 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). You have to use separate timers for each outstanding/unACK'ed packet. Note that at most one timer event can exist in the simulator's event queue at a time for an entity (A or B) but may have many outstanding, unacknowledged packets in the medium, so you'll have to think a bit about how to use this single timer.

You should put your procedures in a file named **rdt_gbn.c.**

Like the earlier version, here also you need to provide a **.doc** file containing sample output from your code. Name the file **output_gbn.doc**. You might want to hand in output 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 annotate parts of your document with a highlighter showing how your protocol correctly recovered from packet loss and corruption.

7. Bonus Part

Addition of the following features in your Go-Back-N protocol will earn you bonus marks. Submit your code for bonus in a file named **rdt_gbn_XYZ.c** where X, Y, Z denote the index of the bonus task in the list below. For example, if you have completed only tasks (i) and (iii), the name of your file will be **rdt_gbn_13.c** whereas if you have completed all the tasks, name the file as **rdt_gbn_123.c.** Again, you are also required to submit a sample output file named **output_gbn_XYZ.doc** highlighting relevant events to prove the correctness of your code.

(i) **Bidirectional transfer of messages.** In this case, entities A and B operate as both a sender and receiver. You may also piggyback acknowledgments on data packets (or you can choose not to do so). To get the emulator to deliver messages from layer 5 to your B_output() routine, you will need to change the declared value of BIDIRECTIONAL from 0 to 1.

(ii) **Accumulated/cumulative acknowledgement.** Instead of sending ACK for each received packet, the receiver will try to send a single ACK for multiple received packets. When a packet is received, an ack

timer will be started instead of immediately sending an ACK. When the timer expires, an ACK will be send that will provide a cumulative ACK for all the packets received within this time.

(iii) **Only one retransmission on receipt of multiple NACKs for a packet.** In the specification of Go-Back-N, the sender is required to retransmit all the packets in the current window in case two ACK's have been received for the same sequence number. However, consider the scenario where the sender sends packets 1-7 and the receiver receives packet 1, packet 2 gets lost and packets 3-7 successfully reach the receiver. In this case, the sender will receive a total of six ACK's for packet 1. This will result in **five** retransmissions of the current window which would be a complete wastage if the receiver gets packet 2 from the first retransmission. Hence, as part of this bonus task you are required to retransmit window only once on receipt of NACK for a particular sequence number, e.g., packet 2 in the example above and ignore the other following NACKs.

8. Helpful Hints and the like

- **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 cannot 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 messages.
- **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.

9. Resources on Internet

You will find a few code bases available for similar problem on the Internet. For this very reason, we have made some changes in the original specification to differentiate the problem. Hence, be careful and avoid dumb copy (without changing variable names, data structures, naming convention, logic flow etc.) from those sources. We will run copy checker against all those publicly available codes for this problem.

10. Submission Guideline and Deadline

Create a folder, use student number as the folder's name, e.g., 1505XYZ and copy all the files there. Only submit the source (**.c**) files and output (**.doc**) files. Then zip the folder and upload to Moodle in the assignment submission link that will be opened before the submission deadline.

The deadline of submission for all sections will be **Sunday**, **January 27, 2019** **at 7:00 am.**

For any further query, you may use either of the following modes: (i) communicate in person, (ii) over e-mail to my address: khaledshahriar@cse.buet.ac.bd, (iii) through forum post in Moodle course page; though unlike you, teachers do not get any notification of forum posts but have to check manually. Hence there may be a substantial delay before your queries get acknowledged and may get dropped in the worst case. Any phone communication is strongly discouraged.

11. Reference

This problem specification and accompanying source code has been adapted from the book's companion website. You can get the original description under the following link:
http://media.pearsoncmg.com/aw/aw_kurose_network_3/labs/lab5/lab5.html

Document Version: 1.0

In case any changes are required, the updated document will have a new version number. You are required to always follow the latest version of this document.