CSE322: Computer Networks Sessional
Level: 3, Term: 2 (January 2020)

Offline Assignment 3: Implementing a Reliable Transport Protocol

1. Overview

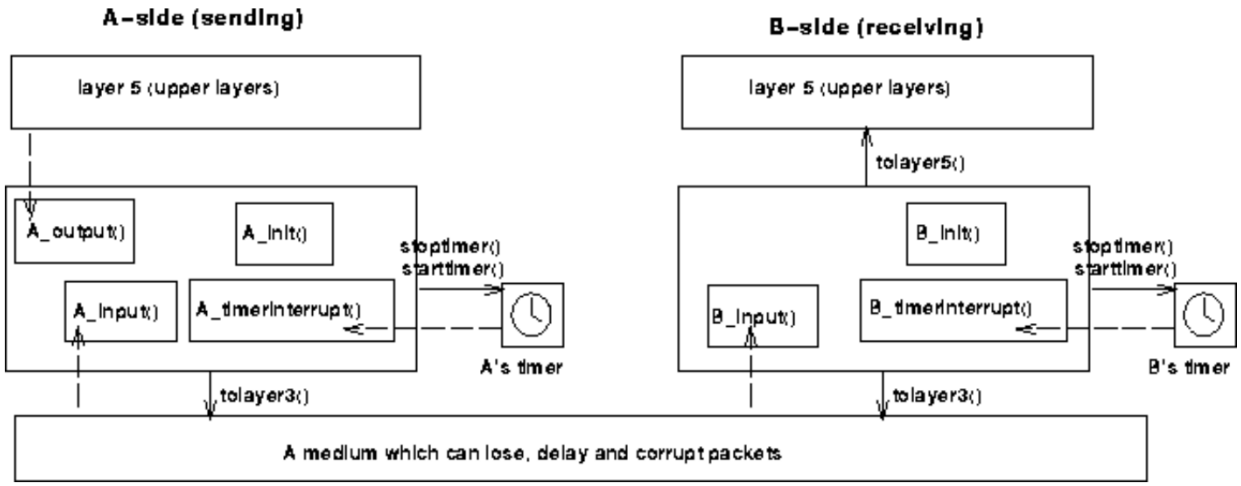
In this laboratory programming assignment, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. You will perform alternating-bit protocol which is quite simple and straightforward. This lab should be **fun** since your implementation 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 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.

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. You will get acquainted with NS-2 soon. 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. (The emulator is already implemented for you)

2. The Routines to Implement

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 protocol 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 protocol to ensure reliable delivery.

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 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 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 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 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 the A-side send) or 1 (for the B-side send), 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 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 values of 0.1 would mean that one in ten packets (on average) are 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) are corrupted. Note that the contents of payload, sequence, ack, or checksum fields can be corrupted. Your checksum should 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 values 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* developers 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 at 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 no 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_1605xyz.c**.

You also need to provide a **.doc** file containing sample output from your code. The name of the file will be **output_1605xyz.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. 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 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 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.

7. 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.

8. Submission Guideline and Deadline

Create a folder, use your student id as the folder's name, e.g., 1605xyz 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 is **Saturday, November 21, 2020 at 2:00 pm**.

For any further query, you may use either of the following modes:

i. E-mail me at sakshar@teacher.cse.buet.ac.bd.
ii. Through forum post in Moodle course page

Any phone communication is strongly discouraged.

9. 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:

https://media.pearsoncmg.com/aw/aw_kurose_network_3/labs/lab5/lab5.html