CS 4480: Computer Networks - Spring 2013

Programming Assignment: 2
Reliable Transport Protocol
Due by 11:59pm MST March 18th 2013
Submit through *handin* on Cade

Overview

In this programming assignment,[1] you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. You have to implement two versions of a reliable transport protocol, namely an Alternating-Bit-Protocol version and a Go-Back-N version. For extra credit you can also implement a Selective Repeat version of the protocol.

A real transport protocol would typically be implemented in the kernel of an operating system. To simplify your development your code will 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 operating system. Stopping/starting of timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

We have C and Java versions of the simulation framework available and you will have to use either of these frameworks and languages to complete this assignment. The description below assumes the C-version of this framework. Notes on using the Java version is at the end of the document.

Assignment details

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 provided by the simulation framework which emulates a network environment. The overall structure of this framework is shown in Figure 1.

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 simulator 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:

[1] Credit: This programming assignment is essentially the same as the *Implementing a Reliable Transport Protocol* described in Kurose and Ross and also makes use of the network simulation framework provided by the authors. The Java version of the network simulation framework has been credited to Sneha Kasera. The description of the assignment also benefitted from versions due to Ibrahim Matta at Boston University and from Keith Labbe from the US Naval Academy.

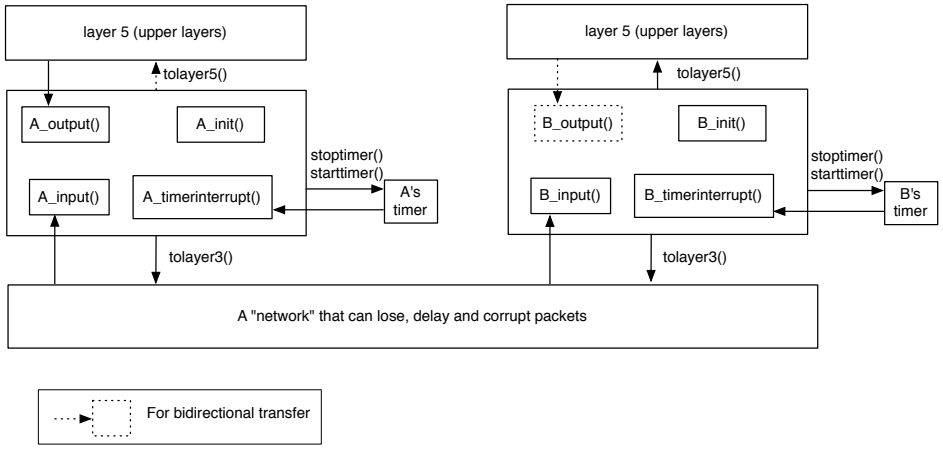

Figure 1: Simulation framework

```
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 discussed in Chapter 3.

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 can 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.

Your program should also gather and report statistics (e.g., number of original data packets transmitted, number of retransmissions, number of ack packets, number of corrupted packets received, average RTT, etc).

You should put your procedures in a file called prog2.c. The initial version of this file is provided with the programming assignment. The provided framework should compile on any OS with a C compiler. It makes no use of UNIX features. (You can simply copy the prog2.c file to whatever machine and OS you choose). **Note, however, that your assignment will be evaluated on the Cade linux environment.**

Software Interfaces

The procedures described above are the ones that you will write. The simulation framework provides 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.

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 together with the simulation framework, and run the resulting program, you will be asked to specify values regarding the simulated network environment:

- **Number of messages to simulate**. The simulator (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 simulator 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 simulator (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 simulator-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.

The Alternating-Bit-Protocol Version

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 is 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.

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.

Output. For your 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 should 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 should annotate your output showing how your protocol correctly recovered from packet loss and corruption.

Your should also show the statistics for your program under various loss and corruption probabilities. You should argue/show whether or not the loss and corruption values for each run (it should be long enough, say, with 1000 messages) are reasonable.

(Hint: if the loss probability is .1, then roughly 10% of the packets should have been lost. If the corruption probability is .15, then roughly 15% of the NON LOST packets should have been corrupted.) If your loss and corruption probabilities were not accurately simulated, your program may have a bug.

The Go-Back-N Version

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. Your protocol should use both ACK and NACK messages.

We would STRONGLY recommend that you first implement the Alternating Bit Version and then extend your code to implement the Go-Back-N Version. 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. You will 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 acceptable 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.

Output. You should 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 received ACKs 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 should annotate your output showing how your protocol correctly recovered from packet loss and corruption.

Your should also show the statistics for your program under various loss and corruption probabilities. You should argue/show whether or not the loss and corruption values for each run (it should be long enough, say, with 1000 messages) are reasonable.

Select Repeat Version (Extra Credit)

Implement a unidirectional transfer of data using Selective Repeat and selective acks (rather than cumulative acks as used in the Go-Back-N part.)

Output. Submit traces, statistics, etc.,that clearly demonstrates your code performs correctly. It is your responsibility to provide output that convincingly demonstrates that your code works correctly (similar to what was asked in the Go-Back-N part of the assignment). Make sure you demonstrate the differences in behavior between Select Repeat and Go-Back-N by appropriate annotations of your output.

Your should also show the statistics for your program under various loss and corruption probabilities. You should argue/show whether or not the loss and corruption values for each run (it should be long enough, say, with 1000 messages) are reasonable.

Grading and evaluation

What to hand in

You should submit your completed assignment electronically via `handin` on Cade by the due date. You submission should consist of a **single** tarball file which contains the following:

1. All the source code for your assignment with inline documentation.
2. A readme.txt file explaining how to compile and/or run your program(s). **Your program should be capable of running all versions of the protocol that you have implemented. E.g., you may select which version to run via a command line argument.**
3. A design.pdf document that describes the program design, how it works and any design tradeoffs considered and made. You should also clearly indicate whether you have implemented the extra credit functionality.
4. A test.pdf document that describes the tests you executed to convince yourself that the program works correctly. Also document any cases for which your program is known not to work correctly.

5. An output.pdf document that shows output that illustrates the correct functioning of your program. You should follow the output directives for each version of the protocol as described above. **Based on your results, you should also compare and comment on the relative performance of each version of the protocol.**

Electronic Submission - Turning in assignments using handin on Cade

Your submission tarball must be submitted on CADE machines using the handin command. To electronically submit files while logged in to a CADE machine, use:

`% handin cs4480 assignment_name name_of_tarball_file`

where `cs4480` is the name of the class account and `assignment_name` (pa1, pa2, or pa3) is the name of the appropriate subdirectory in the handin directory. Use pa2 for this assignment.

Grading

Criteria	Points
Correctly functioning Alternating-Bit Version	30
Correctly functioning Go-Back-N Version	45
Inline documentation	5
Exception handling	5
Design document	5
Thoroughness of evaluation and comparison	10
Total	100
Extra credit functionality	10
Total with extra credit functionality	110

Other important points

- Every programming assignment of this course must be done individually by a student. No teaming or pairing is allowed.
- Your programs will be tested on CADE Lab Linux machines. You can develop your program(s) on any OS platform or machine but it is your responsibility to ensure that it runs on CADE Lab machines. You will not get any credit if the TA is unable to run your program(s).
- Make sure you read the "helpful hints" below.

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).
- **State.** 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 simulator 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 simulator. The simulator 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.
- **QA.** If you are interested in looking at questions Kurose and Ross received (and answered) based on their use of this assignment, check out `http://gaia.cs.umass.edu/kurose/transport/programming_assignment_QA.htm`

Notes on using Java version of simulator

Files to use the Java version of the simulation framework is included in the assignment. NetworkSimulator is an abstract class that is the bulk of the simulator. StudentNetworkSimulator is the only class that you will have to modify; there are unimplemented method which you have to implement. Packet, Message, Event, and EventListImpl are support classes. EventList is an interface. Project is the "driver" for the whole thing. StudentNetworkSimulator.java contains inline comments documenting the interfaces of the other classes that you will need.

The notes given for the C version essentially remains the same for the Java version. The names of methods are the same as C, without "_" in the names.

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

```
public class Packet
{
    private int seqnum;
    private int acknum;
    private int checksum;
    private String payload;
}
```

There are associated useful methods to manipulate a Packet object and those are detailed in the comments of the StudentNetworkSimulator.java class. The packet fields will be used by your protocols to insure reliable delivery.

Your task is to write the java code that simulates a transport protocol that sits on an unreliable network layer. In StudentNetworkSimulator.java class this can all be accomplished by filling in the missing code from the following methods: aOutput(Message message) and bInput(Packet packet). You will simply need to use the previously mentioned software interfaces to turn a message into a packet, send it through layer 3, then retrieve the message on the receiver side and pass it up to layer 5.

When creating the packet for this first task, use a value of zero in each of the data fields int seq, int ack, int check. It is highly encouraged that you use System.out.println to display what data the message is comprised of.