Electrical, Computer, and Systems Engineering
ECSE 4760: Computer Communication Networks (CCN)

September 18, 2001

Laboratory 1: Implementing a Reliable Transport Protocol
Due Friday, October 5th
(Due Wednesday, October 10th, ONLY for tape-delayed students)

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 are two versions of this lab, the **Alternating-Bit-Protocol** (a.k.a "stop-and-wait") version and the **Go-Back-N** version. 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. Stopping/starting of timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

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 which emulate a network environment.

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

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

This declaration, and all other data structure and emulator 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 (i.e. the application layer); your receiving entity should deliver 20-byte chunks of correctly received data to layer5 (i.e. application layer) 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 have studied in 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. The relationship between some of these routines is shown in Figure 1.

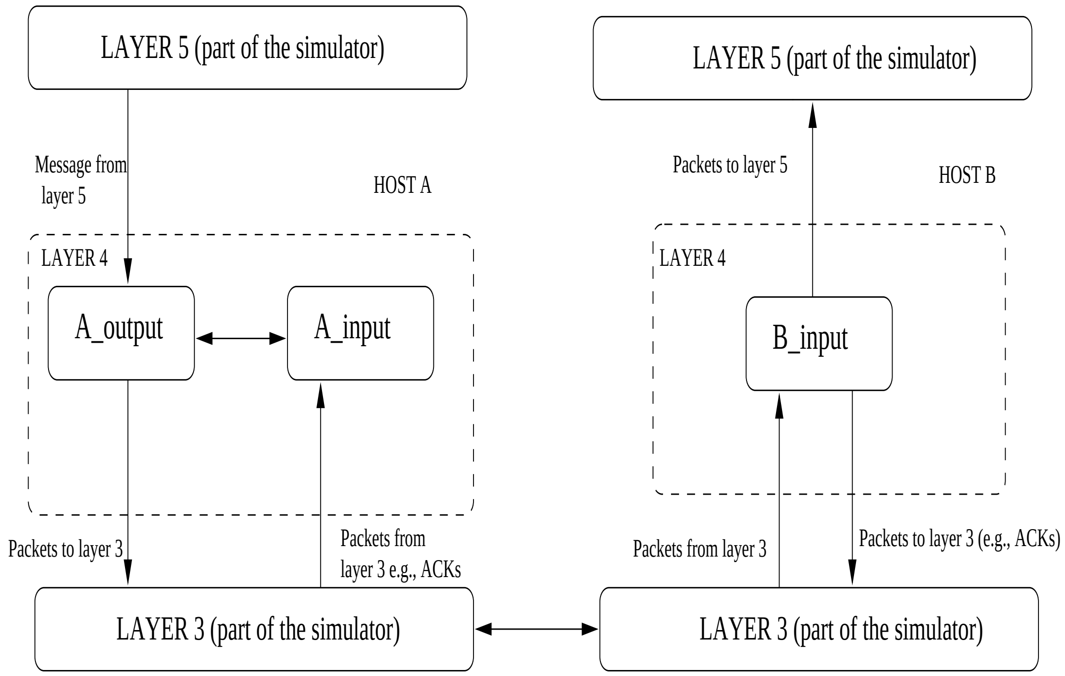

Figure 1: Relationship between various routines

- **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. 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.

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 pre-defined 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 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.

- *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 the 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 be arriving to your sender.

5 The Alternating-Bit-Protocol

For the Alternating-Bit-Protocol part of the lab, 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.

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. We would 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 prog2.c. You will need the initial version of this file, containing the emulation routines we have writen for you, and the stubs for your procedures. You can obtain this program from `http://gaia.cs.umass.edu/kurose/transport/prog2.c`.

This lab can be completed on any machine supporting C. **PLEASE SEE SECTION 8 FOR PLATFORM SPECIFIC DETAILS**.

Make sure you read the "helpful hints" for this lab in the online version of the book. The deliverables for this part of the lab and next are described in Section 9.

6 Go-Back-N Protocol

For the Go-Back-N protocol part of the lab, 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. Consult the alternating-bit-protocol version of this lab above for information about how to obtain the network emulator.

We would **STRONGLY recommend** that you first implement the easier lab (Alternating Bit) and then extend your code to implement the harder lab (Go-Back-N). 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 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.

  The deliverables for this part of the lab and the previous part are described in Section 9.

7 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).

- *Global 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.

- *Time variable:* 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.* 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 Section 8.

8 FAQ and other instructions

- **Frequently Asked Questions:** The authors of the text book have posted a FAQ at: `http://gaia.cs.umass.edu/kurose/transport/programming_assignment_QA.htm`
- **Random Number routine** The setting for the variable `mmm` in the routine `jimsrand` is platform dependent. For **Solaris 2.6, Windows 9x** and **HP-UX** it should be set to 32767. For **Linux 2.2.16+** and **FreeBSD 3.4+** it should be set to 2147483647. This is basically the value of the maximum random number generated by the `rand` function on that platform. This is usually found in the `RAND_MAX` variable defined in `stdlib.h`.

9 Deliverables/Submissions

NOTE: ALL SUBMISSIONS MUST BE DONE VIA WEBCT DROPBOX ONLINE.

The lab will count for 10% of your final grade. 4% is for the alternating bit version and 6% is for the go-back-N version. Partial credit may be awarded only if the code compiles correctly and evidence of progress in the lab is seen in the code.

You will need to submit:

- **Source code**. The C file containing the routines you have written, appropriately commented. Note that your file **MUST** compile to qualify for any credit.
- **A text file containing the output:** Insert `printf` statements at appropriate places in the code to obtain a text file (sample below) which outlines the way packets have been transmitted and received in different loss settings. You should submit output for a run that was long enough so that at least 20 messages were successfully transfered from sender to receiver (i.e., the sender receives ACK for these messages) transfers in each case.
- **Different cases to be considered:** You will need to run you program under the following settings:
  - Zero probability of loss and packet corruption
  - 0.1 probability of loss, zero probability of corruption
  - 0.1 probability of loss and corruption.
  - Trace level set to 1 for all runs of the program.

Please direct all technical questions regarding the lab to the bulletin board. The TAs and instructors will be able to help you in conceptual matters regarding the lab, but will not be available to debug your code.

Sample Alternating-Bit Protocol output

```
-----  Stop and Wait Network Simulator Version 1.1 --------
Enter the number of messages to simulate: 20
Enter  packet loss probability [enter 0.0 for no loss]:0.1
Enter packet corruption probability [0.0 for no corruption]:0.0
Enter average time between messages from sender's layer5 [ > 0.0]:100.0
Enter TRACE:1
 A: Sending message# 0
```

```
 B: Received message# 0
 Recd ACK for message# 0
...
...
 A: Sending message# 10 TOLAYER3: packet being lost
 A: Timed out, retransmitting message# 10
 B: Received message# 10
 Recd ACK for message# 10
...
...
 A: Sending message# 19 Simulator terminated at time 2254.843262
 after sending 20 msgs from layer5
```

Sample Go-Back-N output

```
-----  Stop and Wait Network Simulator Version 1.1 --------
Enter the number of messages to simulate: 20
Enter  packet loss probability [enter 0.0 for no loss]:0.1
Enter packet corruption probability [0.0 for no corruption]:0.0
Enter average time between messages from sender's layer5 [ > 0.0]:10.0
Enter TRACE:1
 A: Sending message# 0
 B: Received message# 0
 A:Recd ACK for message# 0
 A: Sending message# 1
 A: Sending message# 2 TOLAYER3: packet being lost
 B: Received message# 1 TOLAYER3: packet being lost
 A: Sending message# 3
 B: Discarding duplicate or out of order message# 3
 A: Sending message# 4
 A: Timed out, retransmitting message# 1
 A: Timed out, retransmitting message# 2
 A: Timed out, retransmitting message# 3
 A: Timed out, retransmitting message# 4
```

```
 B: Discarding duplicate or out of order message# 4
 A:Recd ACK for message# 1
 A: Sending message# 5
 A: Sending message# 6
 B: Discarding duplicate or out of order message# 1
 B: Received message# 2
 B: Received message# 3
 A:Recd ACK for message# 2
 A: Sending pending message# 7
 B: Received message# 4
...
...
...
 A: Sending pending message# 17
 B: Received message# 13 TOLAYER3: packet being lost
 B: Received message# 14
 B: Received message# 15 Simulator terminated at time 176.059113
 after sending 20 msgs from layer5
```