Project 2: Implementing a Reliable Transport Protocol: Go-Back-N

Developed by Professor Jim Kurose, University of Massachusetts, Amherst

Overview

In this project, you will write the sending and receiving transport-level code for a reliable data transfer protocol: Go-Back-N. 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 environment. Stopping/starting of timers is also simulated, and timer interrupts will cause your timer handling routine to be activated.

Go-Back-N

Go-Back-N is a sliding window protocol where receive window size RWS=1, i.e., the receiver does not accept out-of-order packets. In Go-Back-N, if a timeout occurs for a packet with sequence number n, then the sender retransmits packet n and every outstanding packet in the send window whose sequence number is greater than n. Sender and receiver algorithms for Go-Back-N can be found in Lecture 8's slides.

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 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 overall structure of the emulated environment is shown in Figure 1. In the figure, layer 5 represents the application layer, layer 4 represents the transport layer, and layer 3 represents the network layer.

The unit of data passed between the upper layer (layer5) and your protocol 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, **gbn.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:

```
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 ensure reliable delivery.

The routines you will write are detailed below. 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 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 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.

Software interfaces

The procedures described above are the ones that you will write. The following routines are provided to you and 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 and the procedures provided to you 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 corruption probability. A value of 0.2 would mean that one in five packets (on average) are corrupted. Note that the contents of payload, seqnum, acknum, or checksum fields can be corrupted. Your checksum should thus include the data, seqnum, and acknum fields.
- **Average time between messages from sender's layer5.** You can set this value to any positive value. Note that the smaller the value you choose, the faster packets will be be arriving to your sender.
- **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 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!

The Go-Back-N 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 Go-Back-N unidirectional transfer of data from the A-side to the B-side. The send window size should be set to 8.

You should put your procedures in a file called gbn.c. You will need the initial version of this file, containing the emulation routines that have been written for you, and the stubs for your procedures. You can obtain this program here.

Some considerations for your Go-Back-N code are:

- **A_output(message),** where `message` is a structure of type `msg`, containing data to be sent to the B-side.

  You will have to buffer packets that have been transmitted but not yet acknowledged. 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. In this case, you should buffer the message and send it later when the window is no longer full. 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.

Submitting your project

You must submit your project electronically from pyrite or popeye. Use command "turnin cs586 Project2 src_folder" to turn in your project, where src_folder is the folder that contains gbn.c. You should only submit gbn.c.

You should also hand in a sample output. For your sample output, your procedures should 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 that was long enough so that at least 10 messages were successfully transferred from sender to receiver (i.e., the sender receives ACK for these messages). You should use a loss probability of 0.1, a corruption probability of 0.1, a mean time between arrivals of 10, and a trace level of 2. You should annotate parts of your printout with a colored pen showing how your protocol correctly recovered from packet loss and corruption.

You can use the '`script`' Unix command to record a script of your interaction with the Unix system. Once it's started, the script session is dumping everything that shows up on your screen into some file. To obtain a copy of your program

run, first issue the command '`script`' to the Unix shell; then run your program. If you don't provide a file name to the `script` command, it places its output in a default file named '`typescript`'. Do not use the name of your program's source code file as the filename for the output of the script command. If you type 'script gbn.c', you will overwrite and destroy whatever used to be in gbn.c. When your program terminates, exit from the scripting session by issuing the command 'exit' to the Unix shell. You should hand in a printout of the typescript file.

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 you to compute a sum S, which is the sum of the (integer) `seqnum` and `acknum` 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). Then let S be the checksum.
- You can assume the sequence number space is infinite. That is, the sequence number will never wrap around.
- 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. Then set one of these two probabilities to non-zero, and finally set both to 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.

Grading Guideline

Total Score: 100 points

Sender: 55 points
    A_init(): 5 points
    A_input (): 20 points
    A_timerinterrupt(): 5 points
    A_output(): 25 points
Receiver: 25 points
    B_init(): 5 points
    B_input() : 20 points
Documentation: 5 points (You should add plenty of comments in your code.)
Annotated sample output: 15 points.