Programming Assignment 2:
Implementing a Reliable Transport Protocol
Due April 9th 23:55

This project should be done in pairs.

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 using Go-Back-N and a packet version of TCP. This lab should be fun since your implementation will differ very little from what would be required in a real-world situation.

Since we 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 similar 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.

The routines you will write

The procedures you will write are for the sending entity and the receiving entity. Only unidirectional transfer of data (from sender to receiver) is required. Of course, the receiver will have to send packets to the sender 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 that I provide which emulate a network environment.

The unit of data passed between the upper layers and your protocols is a message, which is defined in the Message class (it is basically a String). Your sending transport entity will thus receive data messages from sender application; and your receiving entity should deliver them correctly to the receiver application.

The unit of data passed between your routines and the network layer is the packet, which is defined in the Packet class.

Your routines will fill in the payload field from the message data passed down from the sender application. The other packet fields will be used by your protocols to ensure reliable delivery, as we've seen 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.

For the sender's transport:

- sendMessage(Message msg) where message contains data to be sent to the B-side. This routine will be called whenever the upper layer at the sending side 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.
- receiveMessage(Packet pkt) where packet is a structure of type pkt. This routine will be called whenever a packet sent from the receiver arrives at the sender. pkt is the (possibly corrupted) packet sent from the sender.
- timerExpired() This routine will be called when the sender'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.
- initialize() This routine will be called once, before any of your other sender routines are called. It can be used to do any required initialization.

For the receivers transport layer you will write:

- receiveMessage(Packet pkt) This routine will be called whenever a packet sent from the sender arrives at the receiver. packet is the (possibly corrupted) packet sent from the sender.
- initialize() This routine will be called once, before any of your other receiver routines are called. It can be used to do any required initialization.

For the packet class:

- setChecksum() sets the checksum field to have a valid value
- isCorrupt() uses the checksum field to check if the packet is corrupt or not.

Software Interfaces

The procedures described above are the ones that you will write. I have written the following routines which can be called by your routines:

Timeline class:

- startTimer(int increment) increment is an *int* value indicating the amount of time that will pass before the timer interrupts. 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() stops the timer if it already running.

NetworkLayer:

- sendPacket(Packet pkt, int to) where pkt is the Packet you wish to send and to is who the message should be sent to (Event.SENDER or Event.RECEIVER) Calling this routine will cause the packet to be sent into the network, destined for the other entity (where it might be lost or corrupted).

ReceiverApplication:

- receiveMessage(Message msg) Calling this routine will cause data to be passed up to layer 5 at the receiver (where it will simply print it out).

Functionality:

- Your final protocol should be able to send a message reliably from sender to receiver under conditions of both loss and corruption.
- You do not need to implement a handshake, assume that it has already taken place and that the first seq num is 0.
- Your program should be able to work with different window sizes.
- In addition to regular Go-Back-N you should implement a simple version of TCP with the following characteristics:
    - Use packet numbers (like go back N) vs. byte numbers.
    - Buffer out of order packets.
    - Only retransmit first packet when timeout occurs
    - Fast retransmit after 3 duplicate acks
    - Window size does not need to change

The Simulated Network Environment

A call to method sendPacket(Packet pkt, int to) sends packets into the medium (i.e., into the network layer). Your methods receiveMessage(Packet pkt) 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 run the resulting program, you will be asked to pass values regarding the simulated network environment:

- Input Filename with messages. Each line in the file is a message and will be sent at a random time given the following parameters.
- Average time between messages from sender's application layer. 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.
- Loss Probability. 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 Probabilities. 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.
- Window Size. This should allow you to control the window size for the go-back-n protocol.
- Go-Back-N vs. TCP. A parameter which sets if you are using TCP or go-back-n. 0 means you are using go-back-n, 1 means you are using your simple TCP
- Tracing. Setting a tracing value of 1,2 or 3 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 of value 1 prints out times for sending, receiving and timers expiring events. A tracing of value 2, prints out when a message is corrupted and lost. A tracing value greater than 2 will display messages that are related to the event timeline. You probably will not need to use that one.

Your Implementation

You are to write the procedures which together will implement a Go-Back-N unidirectional transfer of data from the sender to the receiver.

Submission:

Besides your code (which should be well commented), please submit the following report:

- A short introduction to what the program is trying to achieve.
- A description of the both reliable data transfer protocols you implemented. This should include a full explanation of the protocol such as: the expected exchange of messages, rules about how the window changes, what to do in a timeout, etc.
- A description of your code design and choice of data structures.
- Correctness Results - In this section you should present a simple print out of your program sending and receiving at least 5 messages (with at least one loss and one corruption), and draw a sending diagram which shows that the time it took to send the 5 messages is correct according to your calculation. You should show this for both TCP and go-back-n.
- Results Analysis - Show a comparison of the different algorithms when changing different parameters. Make sure to present some graphs which show how the time depends on different parameters.
- Conclusion - Mention which algorithms work the best and under what conditions.
- References - If necessary.
- A section on what each member of the team did as part of the project.

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).
- 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 you're debugging your procedures.

- You can assume that the three way handshake has already taken place. You can hard-code an initial sequence number into your sender and receiver.
- The program ends after an ack for the last packet arrives successfully at the sender.