Programming Assignment #2

ELEN E6761 - Computer Communication Networks — Due 10/3 4:10pm (CVN: 10/6 5pm)
Fall 2000 — Prof. Rubenstein

Goal

In this programming assignment, you will build a reliable transfer protocol on top of UDP. You will design both a sender and receiver side participant: the **sender** is the host that starts with a file - this file is to be delivered to the **receiver**. Note that both the sender and receiver will be sending and receving packets (the sender sends data and receives acknowledgments, the receiver receives data and sends acknowledgments).

You will design two versions: in one version, you will use Go-back-N style recover, and in the other, the style of recovery will be Selective Repeat. In both versions, the sender and receiver will need to perform connection setup, connection teardown, as well as connection termination after a significantly long timeout. The connection establishment and teardown mechanisms can be the same for both parts of the assignment.

wrapped_sendto

In both parts of the assignment, on the both the sender and receiver side, you should replace any call you would normally make to sendto with a call to wrapped_sendto with an identical parameter list. wrapped_sendto is a function that will be provided to you, written specifically for this assignment. It emulates various styles of packet drops and reordering of packets so that you can test the thoroughness of your reliability protocol. This function is needed because chances are, network loss rates on the real network are so low that not emulating packet losses would make the code of little interest, especially if both of your hosts reside on the same LAN. See the course materials web-page for the code patch that contains the wrapped_sendto function. You can simply place #include "wrapped_sendto.c" in the body of your code after you have included the standard header files.

Requirements of your reliable transfer protocol

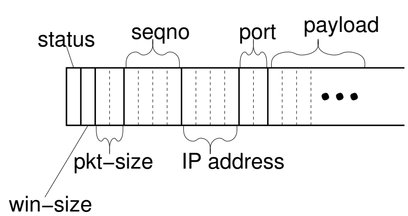

Figure 1: Byte structure for your packets

The packets that you transmit should have the structure shown in Figure 1:

- byte 1: packet status, set to either:

  0: connection setup

  170: data (bit-sequence 10101010)

  85: acknowledgment (bit-sequence 01010101)

  255: connection teardown

- byte 2: size of the session transmission window

- bytes 3-4: size of packet (in bytes), including the header bytes
- bytes 5-8: sequence number
- bytes 9-12: IP address where packet originated
- bytes 13-14: port of origin
- bytes 15 on: payload

Note that if the payload is n bytes long, then the size entered in bytes 3 and 4 should equal $n + 14$. Your protocol should be designed to handle / do the following:

- protocols should handle arbitrary packet losses of data and/or ACKs
- protocols should handle packet reordering of data and/or ACKs
- The sender should not send **new** packets faster than a rate of 10 per second (i.e., you must regulate the rate at which it sends new packets - retransmissions do not need to be controlled. If you feel that it is easier to control the aggregate rate of new packets and retransmissions, you may do so instead). Hint: use `select()` to regulate the rate of flow.
- if acknowledgments do not arrive within a specified period of time, t, then the sender should automatically retransmit (t should be no more than a second.
- if there is no communication from the other host after a period of time T where T is much larger than t (e.g., 5 or 10 times as large), then the host should tear down the connection and report a failure to complete the transfer to the application.
- half-duplex data transfer (full duplex would be both hosts sending data to one another).
- The receiver should initiate the request for a connection.
- The sender and receiver should both verify that the address and port contained in the packet matches the IP address and port that are returned by the call to `recvfrom()`.

Note that the introduction of timeouts will require the use of the `select()` call. i.e., if both the sender and receiver block on their respective calls to `recvfrom()` (the receiver waiting for data and the sender waiting for ACKs), the protocol will hang at both ends if all packets in the current window are lost during transmission.

Part 1: Go-Back-N

You can treat out-of-order packets as either lost packets, or you can buffer them, as long as they fit within the current receiver window.

Part 2: Selective Repeat

You should buffer packets that arrive out of order, so long as they fit within the receiver window.

You can choose the format of the acknowledgment (e.g., a bit-vector, a string of bytes indicating the packets received, each ACK packet contains the acknowledgment for a single packet, etc.)

Questions

- Sometimes (but almost never), packets are **fragmented**, i.e., only part of a packet is retrieved on a call to *recvfrom()*, and additional calls must be made to retrieve the remainder of the packet. How would you have to modify your protocol design to handle fragmentation?
- Recall that the accept() function call used with TCP sessions spawns a new socket for data transfer after connection setup. How would you have to modify your sender and receiver protocol to allow the sender to handle multiple receivers at one time in a similar fashion?

What you should turn in

You should submit both a hard-copy, as well as an on-line copy. Before the due date, we will point you to a web page that will give detailed instructions of how / where to send your submission.

Your submission should include:

- your code (4 files: Go-back-N sender, Go-back-N receiver, Selective Repeat sender, Selective Repeat receiver)
- A description of your design decisions. Many of the details of the design of the protocols are open. Explain what you did and why
- A few sample runs of your code. We will put out a list next week of mandatory scenarios that you should include (i.e., various ways to configure the wrapped_sendto() function, as well as the format of the output.
- answers to the questions above.