Programming Assignment 2 (50 pts)
Data Link Layer Client and Server Processes
Using Tanenbaum's PAR Protocol

Due: 11:59 a.m. Friday, February 13, 2004

Introduction

This assignment exposes the student to data link layer issues by implementing the PAR (Positive Acknowledgement with Retransmission) protocol on top of an emulated physical layer ***{real TCP does the actual transmissions for your physical layer}***. Assignment 2 is a major step towards a more complete server implementation in assignment 3.

The assignment is to build two processes (a **client** and a **server)** that communicate at the data link layer. Both processes *send* and *receive* frames. The responsibilities of these two processes include: byte stuffing, error detection and the PAR protocol with a timeout mechanism that causes a frame retransmission when frames are not promptly acknowledged.

Assignments 2 and 3 are to be completed in two person teams. **To select your own partner, both team members must send an email indicating both team members to** cs4514-ta@cs.wpi.edu **by noon. Friday, January 30th .** All team assignments will be announced by **Monday, February 2nd**.

Frame Format

Information at the data link layer is transmitted between the client and the server in frames. All frames must have two framing bytes **[hex '7E']** (one at each end of the frame), one byte for the sequence number, and one byte for error-detection. The client process sends data frames that contain from 1 to 72 bytes of payload (encapsulated data from the network layer). Data frames will also contain an end-of-packet byte. The server process sends only acknowledgement (ACK) frames consisting of the two framing bytes, **zero** bytes of payload, the sequence number byte, and the error detection byte. However, all bytes **inside** both frame types are subject to byte stuffing.

Client Process

The client process emulates the lower three OSI layers (network, data link, and physical layer).

The command line for initiating the client process is:

client *filename servername*

where

filename indicates that the input file for packets is *filename.raw*

and

servername indicates the logical name for the server machine (e.g., ccc4.wpi.edu)

The ***client network layer*** reads from the input file. The input file contains a series of *simulated* IP packets with a 30 byte minimum size and a 230 byte maximum size. The input file contains information in the following "raw" format:

p an integer specifying the number of packets in the file

This is followed by **p** packets in the following format:

len an integer specifying the number of bytes in the **ith** packet
packet the packet in *raw byte* form

Initially, the client process calls the physical layer to establish a connection with the server. Once a connection has been established, the client network layer reads in one packet at a time from the input file. The client network layer sends a packet to the client data link layer.

The ***client data link layer*** receives packets from the network layer and converts them to frames. The client data link layer gives frames to the client physical layer to be *sent* via TCP. Upon receiving each packet from the network layer, the client data link layer splits the packet into payloads. The data link layer builds each frame as follows:

- put the payload in the frame
- deposit the proper contents into the end-of-packet byte to indicate if this is the last frame of a packet
- compute the value of the error-detection byte and put it in the frame
- *byte-stuff* all of the above bytes
- start a timer
- send the frame (including the framing bytes) to the physical layer.

The ***client physical layer*** sends the constructed frame as an actual TCP message to the physical layer of the server process.

The ***client data link layer*** then waits to *receive* an ACK frame. If the ACK frame is received successfully before the timer expires, the client sends the next frame of the packet or gets the next packet from the network layer. If the ACK frame is received *in error*, record the event in the log **and continue the data link layer as if the ACK was never received.** If the timer expires, the client retransmits the frame.

The client records significant events in a log file *client.log*. Significant events include: packet sent, frame sent, frame resent, ACK received successfully, ACK received in error, and timer expires. For logging purposes identify the packet and the frame within a packet by number for each event. Begin counting packets and frames at 1.

When all the packets have been sent, the client closes the connection to the server and terminates.

Server Process

The server process emulates the same three layers as the client process (network, data link and physical layer). The server process is always started first.

The command line to start the server is simply:

server

where

server.out indicates the name of the file where the server writes out packets

and

server.log indicates the file which records significant server events.

The server begins by waiting for the establishment of a connection from a client. Once the connection is established, the ***server data link layer*** cycles between *receiving* a frame from the server physical layer, assembling the packet and possibly sending the packet up to the server network layer, and *sending* an ACK frame back to the client via the server physical layer. There is no need for a timer at the server. **Note: the end-of-packet byte is used to indicate to the server data link layer the last frame of a packet.** When the client closes the connection to the server, the server terminates.

The ***server data link layer*** has to ***unstuff*** frames and check for an error using the **error-detection** byte. If the received data frame is in error, the server records the event and waits to receive another frame from the client. The server data link layer checks received frames for duplicates and reassembles frames into packets and sends one packet at a time to the ***server network layer*** where they are written to *server.out*. Note – the server needs to send an ACK when a duplicate frame is received due to possibly damaged ACKs. The server records significant events including frame received, frame received in error, duplicate frame received, ACK sent, and packet sent to the network layer in *server.log*.

Frame Error Simulation

Since real TCP guarantees no errors at the emulated physical layer, you must inject artificial transmission errors into your physical layer.

Force a **client transmission error** in every 5^{th} frame sent by flipping any single bit in the error-detection byte prior to transmission of the frame. Force a **server transmission error** every 6^{th} ACK frame sent by using the same flipping mechanism. (i.e., frames 5, 10, 15, … sent by the client will be perceived as in error by the server and ACK frames 6, 12, 18, … sent by the server will be perceived as error by the client.) When the client times out due to either type of transmission error, it resends the same frame with the correct error-detection byte.

Assignment Hints

- **[DEBUG]** Build and debug your programs in stages. Begin by getting the client and server to work without errors and without the timer. Then add the error generating functions and the timer mechanism on the client. Get the assignment working on a single machine first. When it is all working on one machine, move the client and server processes to separate machines prior to turning in the assignment. **Note -** Make sure your server runs on **one of the CCC Linux machines**!
- **[Error-Detection]** While **CRC** at the bit level will be discussed in class, I recommend using a byte-by-byte **XOR** of all the internal bytes for creating your error-detection byte. For the ACK frame, the error-detection byte then becomes simply a copy of the sequence number byte.

- The **correct** way to handle a timer and an incoming TCP message **requires** using a timer and the ***select*** system call. You will lose points if you use polling to do this assignment.

- **[Performance Timing]** You **must** measure the total execution time of the complete emulated transfer and print this out in file *client.log*.

- **[Timer] The protocol implemented can fail if there is a *premature timeout.* Set the timeout period large enough to insure no premature timeouts.**

- **port numbers:** You can "hardwire in " the port numbers for this assignment because there is only one client and one server. However, a more general solution is better preparation for program 3.

- The **actual content of the packets** is not a concern in this assignment except that the packets received by the server at the network layer should **exactly** match the packets sent by the client.

- **[Documentation]** You need to clearly explain your design for the end-of-packet byte in your comments. **Remember: This a team project and all routines must specify the author as part of the documentation!! You CANNOT simply attribute all routines to both team members!!**

Do not wait for the official test data to work on this assignment. Build a simple version of your own test data to get going. The TA will send out a sample program showing exactly how to read the raw input file.

What to turn in for Assignment 2

The TA will make an official test file available a couple of days before the due date. Turn in your assignment using the *turnin* program. Turn in the two source programs *client.c* and *server.c*, the client and server output files corresponding to running the programs using the TA's data, a README file and possibly a make file.