Programming Assignment 2 (15 pts)

Due: 11.59 p.m. Tueday, February 14, 2006

Client Server for downloading MPEG Video files

Using Tanenbaum's 1-bit Sliding WindowProtocol (Protocol 4)

Introduction

The goal of this assignment is to send mpeg video files located at a server to a client. The client and server run on separate CCC Linux machines and communicate at the data link layer by *sending* and *receiving* frames. Both the client and server emulate three OSI layers (application/network, data link and physical layer).

This assignment exposes the student to data link layer issues by implementing a 1-bit sliding window protocol on top of an emulated physical layer ***{real TCP does the actual transmissions for your physical layer}***.

The responsibilities of these two processes include: byte stuffing, error detection and the 1-bit sliding window protocol with a timeout mechanism that causes a frame retransmission when frames are not promptly acknowledged. Since data is sent only from the server to the client, no piggybacking of acknowledgements should be implemented.

THE SERVER

The server should be written to run on any arbitrary CCC Linux machine. The server emulates the lower three OSI layers (network, data link, and physical layer). The server is always started first.

The command line for initiating the server is:

server

The server also creates a log file ***server.log*** that you can use later for debugging.

Server Application/Network Layer

You should place three short mpeg files (about 2-5MB) at the server. The ***server application layer's*** responsibility is to read each of your mpeg files in turn (***filename.mpg***) and send them to the client. The ***server application layer*** indicates to the ***server network layer*** when it has completely read in a video by setting the end-of-video indicator. Note that for this assignment, you do not have to separate the application and network layers.

Initially, the ***client network layer*** calls the physical layer to establish a with the ***server network layer.*** The ***server network layer*** begins receiving 300-byte "chunks" of videos and depositing each 300 byte chunk into a packet payload. Additionally, the packet payload contains one byte as an end-of-video indicator for the application layer. The ***server network layer*** sends the packet to the data link layer and waits for an **ACK** from the ***video client network layer.***

Server data link layer

The responsibilities of the data link layer involve error detection and the 1-bit sliding window protocol with a timeout mechanism that causes a frame retransmission when the frames are not promptly acknowledged.

Frame Format

Information at the data link layer is transmitted between the server and client in frames. All frames need a frame-type byte to distinguish data and ACK frames. All data frames must have two bytes for the sequence number, two bytes for error-detection, and one end-of-packet byte. The client server process sends data frames that contain from 1 to 80 bytes of payload (encapsulated data from the network layer packet). ACK frames consist of **zero** bytes of payload, a two-byte sequence number, and a two-byte error detection field.

The ***server data link layer*** receives packets from the ***server network layer,*** converts packets into frames and sends frames to the ***server physical layer.*** Upon receiving each packet from the ***server network layer,*** the ***server data link layer*** splits the packet into frames 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 bytes and put them into the frame.
- Start a frame timer
- Send the frame to the ***server physical layer***

The ***server data link layer*** then waits to receive a frame. If the received frame is a data frame, then its payload is a network layer ACK packet. The ***server data link layer*** then sends the valid ACK packet up to the ***server network layer*** and then waits to receive a packet from the ***client network layer.*** If the received frame is an ACK frame successfully received before the timer expires, the server sends the next frame of the packet. When the last frame of a packet has been successfully ACK'ed, the ***server data link layer*** waits to receive a data frame. If an ACK frame is received *in error,* this event is recorded in the log **and the server data link layer continues as if the ACK was never received.** If the timer expires, the ***server data link layer*** retransmits the frame.

Server physical layer

The ***server physical layer*** sends the frame received from the ***server data link layer*** as an actual TCP message to the ***client physical layer.*** The ***server physical layer*** receives frames as actual TCP messages from the ***client physical layer.*** This triggers a received frame event from the ***server data link layer.***

The server records significant events in a log file ***client.log.*** Significant events include: packet sent, frame sent, frame resent, ACK frame received successfully, ACK packet received successfully, ACK frame 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 (e.g. "frame 2 of packet 218 was retransmitted").

THE VIDEO CLIENT

The client should also be written to run on an arbitrary CCC Linux machine. The client emulates the lower three OSI layers (network, data link, and physical layer).

The command line for initiating the client is:

client servername

where

servername indicates the logical name for the server machine (e.g., ccc4.wpi.edu). The client communicates with the server through an ephemeral port that you can choose and hardcode in your program. The client also creates a log file ***client.log*** that you can use later for debugging.

Client Application/Network Layer

The ***client application layer*** receives 300-byte chunks of video in the form of network packets. It puts these chunks together to reconstruct the original mpeg file and writes them out to a local directory. The ***client application layer*** interrogates the end-of-video byte in the packet to know when the current packet is the last packet for an mpeg video so the specific mpeg file is closed.

After the ***server application layer*** has processed each packet**,** the server network layer creates an ACK packet and sends it to the server data link layer.

Client Data Link Layer

The ***client data link layer*** initiates a connection with the ***server data link layer***. Once the connection is established, the ***client data link layer*** cycles between ***receiving*** a frame from the physical layer, reassembling the packet and possibly sending the packet up to the network layer, and ***sending*** an ACK frame back to the ***server*** via the ***client physical layer***. The ***client data link layer*** sends ACK frames consisting of two bytes of sequence number and the two error detection bytes. There is no need for a timer at the ***client.*** **Note: the setting of the end-of-packet byte is used to indicate to the client the last frame of a packet.** When the client closes the connection to the server, the server terminates.

The ***client data link layer*** has to check for an error using the **error-detection** byte. If the received data frame is in error, the client records the event and waits to receive another frame from the server. The client data link layer checks received frames for duplicates and reassembles frames into packets and sends one packet at a time to the ***client network layer***. Note – the client needs to send an ACK when a duplicate frame is received due to possibly damaged ACKs. The client records significant events including frame received, frame received in error, duplicate frame received, ACK sent, and packet sent to the network layer in ***client.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 6th ACK frame sent by flipping any single bit in the error-detection byte prior to transmission of the frame. Force a **server transmission error** every 4th data frame sent by using the same flipping mechanism. (i.e., frames 6, 12, 18, … ACKs sent by the client will be perceived as having an error by the server and frames 4, 8, 12, … sent by the server will be perceived as having an 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.

Assume for this assignment that all data frames sent by the ***client data link layer*** are transmitted "error free". Therefore, the ***server data link layer*** does **NOT** need to ACK the data frames sent by the ***server data link layer.***

Hints

- **[DEBUG]** Build and debug your programs in stages. Begin by getting the programs 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 client and server runs on one of the CCC Linux machines. Include the name of the CCC machine that you tested on, in your documentation.

- **[Error-Detection]** While **CRC** at the bit level has been 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. Since each client packet will have an associated timer, you need to design your client to track multiple timers.

- **[Performance Timing]** You **must** measure the total execution time of the complete emulated transfer and print this out in file ***server.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. A more general solution is possible but you do not have to implement it.

- The **actual content of the mpeg video files** received by the client should match the initial mpeg files sent by the server. Differences shall result in loss of points.

- **[Documentation]** You should document all your design choices and all significant aspects of your work.. **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 test mpeg files. Just get three short mpeg files from the web and get going.

What to turn in for Assignment 2

The TA will make an 3 official mpeg files 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, and a README file.