Program 2 **{January 30, 2010}** **45 points**

Sending Photos to a Concurrent Photo Server
Due: Thursday, February 4, 2010 at 11:59 p.m.

Introduction

The goal of this assignment is to send sets of five digital photographs from two or more clients to a photograph gallery server. Both the **Clients** and the **Server** emulate three OSI layers (application/network, data link, and physical layer). The **Clients** and **Server** run on separate CCC Linux computers and communicate by ***sending*** and ***receiving*** frames via their physical layers.

This assignment exposes the student to the concept of network protocol layers by implementing the PAR (Positive Acknowledgement with Retransmission) data link protocol on top of an **emulated** physical layer **{TCP does the actual transmissions for the physical layer}**.

The Clients

The **n Client** programs should run on any arbitrary CCC Linux machine. The command line for initiating the client is:

client id servermachine

where

id identifies the client number chosen and stored as a one byte integer
{make sure ***ids*** are unique within the set of clients per simulated run!!}.

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

and

clientid.log indicates the file which records significant **Client id** events.

The **Clients** communicates with the **Server** assuming knowledge of a unique "well-known" port for the photo server.

The Client application/network layer

C**lient id's** ***application layer*** responsibility is to read five digital photos from files ***photoid1.jpg*** to ***photoid5.jpg*** and send each photo one at a time to the photo server. The ***client application layer*** indicates to the

client network layer when it has completely read in a photo by setting the end-of-photo indicator. **{Note, in this assignment the abstraction of separating these two layers is not necessary}.**

Initially, the ***client network layer*** calls the ***client physical layer*** (**this is an emulation cheat**) to establish a connection with the ***server network layer*** for **Client id**. Once a connection has been established, the ***client network layer*** begins receiving 128-byte "chunks" of photo images from the ***client application layer*** and depositing each 128-byte chunk into a packet payload. Additionally, the packet payload contains one byte to indicate the client id number and one byte as an end-of-photo indicator for the ***server application layer***. The ***client network layer*** sends the packet to the ***client data link layer*** and waits for an **ACK** packet for this **client** from the ***server network layer***. Note an ACK packet includes the client id number.

When the last photo has been sent and all its packets have been ACK'ed, the ***client network layer*** calls ***the client physical layer*** a second time to close the TCP connection to the Server. After all the logged information has been recorded on the ***client*id*.log*** file, the **Client** terminates.

The Client data link layer

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

Frame Format

Information at the **data link layer** is transmitted between **Clients** and the **Server** 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, one end-of-packet byte and two trailing bytes for error-detection. The **Client** 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 ***client data link layer*** receives packets from the ***client network layer***, converts packets into frames and sends frames to the ***client physical layer***. Upon receiving each packet from the ***client network layer***, the ***client data link layer*** splits the packet into frame 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 trailer.
- start a frame timer.
- send the frame to the ***client physical layer***.

The ***client data link layer*** then waits to *receive* a frame. If the received frame is an ACK frame successfully received before the timer expires, the client sends the next frame of the packet. When the last frame of a packet has been successfully ACK'ed, the ***client data link layer*** waits to receive a data frame. If the received frame is a data frame, then its payload is a network layer ACK packet. The ***client data link layer*** then sends the valid ACK packet up to the ***client network layer***, and then waits to receive a new packet from the ***client network layer***.

If an ACK frame is received *in error*, this event is recorded in the log **and the client data link layer continues as if the ACK was never received.** If the timer expires, the ***client data link layer*** retransmits the frame.

Client id records significant events in a log file ***client*id*.log***. Significant events include: packet sent, frame sent, frame resent, ACK frame received successfully, ACK frame received in error, ACK packet received successfully, 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 ***client data link layer*** needs to keep running tallies of significant events. These totals, written to the ***client*id*.log***, include: the total number of frames sent, the total number of frame retransmissions, the total number of good ACK frames received and the total number of ACK frames received with errors.

The Client physical layer

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

The Concurrent Photo Server

The **Server** also should be written to run on an arbitrary CCC Linux machine. The **Server** emulates the same three layers as the client process (application/network, data link and physical layer). The **Server** is always started first.

The command line to start the server is simply:

server

where

photonew*id*1.jpg to ***photonew*id*5.jpg*** indicates the name of the files in the photo gallery written out by the **Server** on behalf of **Client id**.

and

server*id*.log indicates the file which records significant **Server** events handled by the server child process communicating with **Client id**.

A concurrent photo server implies that the parent server process listens for the establishment of a TCP connection from any new client. Once the connection to a new client is established, the server forks a child that participates in all further communications with the **Client id.** Each child server process implements the three emulated protocol layers. When **Client id** closes the connection to the server, the respective server child process terminates.

The Server application/network layer

The ***server application layer's*** responsibility is to take 128-byte photo chunks out of network packets sent by **Client id** to reconstruct that client's five photos and write them out to the files in the photo gallery. The ***server application layer*** interrogates the end-of-photo indicator byte in the packet to determine when the current packet is the last packet of a photo image so the specific photo file can be closed.

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

The Server data link layer

The ***server data link layer*** cycles between *receiving* a frame from the ***server physical layer***, assembling a packet and possibly sending the packet up to the ***server network layer***, *sending* an ACK frame back to **Client id** via the ***server physical layer,*** *receiving* an ACK packet from the ***server network layer*** and *sending* it as a data frame to the ***server physical layer***. The ***server 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 server. **Note - the setting of the end-of-packet byte indicates to the *server data link layer* that the current received frame is the last frame of a packet.**

The ***server data link layer*** checks received frames for duplicates, reassembles frames into packets and sends one packet at a time to the ***server network layer***. Note – the ***server data link layer*** needs to send an ACK frame when a duplicate frame is detected due to possibly damaged ACK's. The **Server** records significant events associated with **Client id** including frame received, frame received in error, duplicate frame received, ACK frame sent, ACK packet sent and packet sent to the network layer in ***serverid.log*** {one log for each child server process}.

The ***server data link layer*** needs to keep running tallies of significant events. These totals, written to the ***servertid.log***, include: the total number of frames received, the total number of frames received in error, the total number of good ACK frames sent, the total of bad ACK frames sent and the total number of ACK packets sent.

The Server physical layer

The ***server physical layer*** receives a frame from the ***client physical layer*** as an actual TCP message and this triggers a received frame event for the ***server data link layer.*** The ***server physical layer*** receives both data frames (namely, ACK packets) and ACK frames from the server data link layer. The ***server physical layer*** sends these frames as actual TCP packets to the ***client physical layer***.

General data link layer issues

The **data link layer** has to check for transmission errors using the **error-detection** bytes. If the received data frame is in error, this event is recorded and the receiving process waits to receive another frame. While **CRC** at the bit level is possible, it is strongly recommended that you use a two-byte **XOR folding algorithm** of all the frame bytes to create your error-detection bytes and to validate frames upon reception. For the ACK frame, the error-detection bytes simply become a copy of the two-byte sequence number.

Simulating frame errors

Since TCP guarantees no errors in the emulated physical layer, your program must inject artificial transmission errors into your physical layer.

Force a **client transmission error** in every **6th** frame sent by flipping any single bit in the error-detection bytes prior to transmission of the frame. Force a **server transmission error** every **8th** ACK frame sent by using the same flipping mechanism. (i.e., frames 6, 12, 18, … sent by the client will be perceived as in error by the server and ACK frames 8, 16, 24, … 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.

Assume for this assignment that all data frames (i.e., ACK packets) sent by the ***server data link layer*** are transmitted "error free" Therefore, the ***client data link layer*** does **NOT** need to ACK the data frames sent by the ***server data link layer*** {**This is a second simulation cheat used to simplify the assignment!**}.

Assignment Hints

- **[DEBUG]** Build and debug your programs in stages. Begin by getting one **Client** and the **Server** working without transmission errors and without a timer. Next, debug the concurrent server with two concurrent clients. Then add the error generating functions and the timer mechanism on the client side. Initially, the **Clients** and **Server** can exist on the same machine if this simplifies your debugging. However, at least one member of the programming team needs to focus on making sure the final project permits the **Clients** and the **Server** to run and be tested by the TA on any arbitrary CCC Linux machine.

- 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 of all the photos for each of the **n** Clients. Be sure to print this result out in readable form as the last item in the file ***client*id*.log***.

- **[Timer] The PAR protocol 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 well-know port number of the **Server** for this assignment.

- The **actual content of the photo files** written by the **Photo Server** should **exactly** match the photo files read by the client.

- **[Documentation]** Several small design decisions are deliberately vague in this assignment. The project team MUST explain all these design decisions both as documentation in the code and via a separate README file. **Remember: This is a team project and all routines must specify only a SINGLE primary author for each routine as part of the documentation!! You CANNOT simply attribute routines to all team members!!**

Do not wait for the official test data to work on this assignment. Use your own digital photos to test out the clients prior to the TA releasing the official test photos.

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 a tarred file using the ***turnin*** program. The tarred file should include the two source programs ***client.c*** and ***server.c***, a **README** file and a ***make*** file that the TA can use to compile, run and test your clients and the concurrent server.