Program 3 **{August 19, 2015}** **60 points**

Sending Photos to a Concurrent Photo Server
Due: Tuesday, October 13 at 4 p.m.

Introduction

The goal of this assignment is to send sets of digital photographs from multiple clients to a photograph gallery server. The clients and server need to be designed to run on separate CCC Linux computers and communicate at the data link layer by ***sending*** and ***receiving*** frames. Both the clients and the concurrent server emulate three OSI layers (application/network, data link, and physical layer).

This assignment exposes the student to network protocol layers by implementing the PAR (Positive Acknowledgement with Retransmission) data link protocol on top of a **emulated** physical layer **{TCP does the actual transmissions at the physical layer}**. This is the ONLY programming assignment in CS3516 to be completed by two-person programming teams.

Photo Clients

Each Photo Client should be written to be run on any arbitrary CCC Linux machine. The command line for initiating the client is:

./client servermachine id num_photos

where

servermachine	specifies the logical name for the server machine (e.g., cccWORK4.wpi.edu).
id	is a unique integer identifying this client.
num_photos	indicates the number of photos this client wants to upload to the Photo Server.

and

client_id.log	indicates the file which records significant events for this client. ***id*** is the ASCII string corresponding to command line argument matches ***id***.

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

The Client Application/Network Layer

The ***client application layer's*** responsibility is to read digital photos from **jpg** photo files and send them one at a time to the photo server. All photos are identified via photo**ij**.jpg where **i** corresponds to the client's id and **j** corresponds to the **j**th photo for the **i**th client. 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, for this assignment the abstraction of separating these two layers is not necessary}.**

Initially, the ***client network layer*** calls the physical layer to establish a connection for this particular client with the corresponding ***server network layer***. Once a connection has been established, the client network layer begins receiving **256 byte** "chunks" of photos from the photo file and depositing each 256 byte chunk into a packet payload. Additionally, the packet payload contains one byte as an end-of-photo indicator for the application layer. The ***client network layer*** sends the packet to the ***client data link layer*** and waits for a network layer **ACK** packet from the ***photo server network layer***.

When the last photo for a client has been sent and ACK'ed, the ***client network layer*** calls ***the client physical layer*** to close the connection to the server and terminate the client.

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 the client 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 bytes for error-detection. The client process sends data frames that contain from **1 to 130 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.
- start a frame timer.
- send the frame to the ***client physical layer***.

The ***client data link layer*** then waits to ***receive*** a frame from the ***server data link layer***. 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.

The Client Physical Layer

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

The client records significant events in a log file ***client_id.log***. Significant events include: packet sent, frame sent, frame resent, ACK frame received successfully, ACK packet received successfully, data frame received in error, 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 ***client data link layer*** needs to keep a running tally of the significant events recorded in ***client_id.log.*** The final entry in the ***client_id.log*** should include: the total number of frames sent, the total number of frame retransmissions, the total number of good ACKs received and the total number of ACK frames received with errors. Note – a very useful debugging tool is to be able to print the running tally BEFORE the client has finished.

The Concurrent Photo Server

The concurrent Photo Server 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). However, as the concurrent server handles multiple client conversations, it maintains separate versions of these three layers for each currently active client. The concurrent photo server is always started first.

The concurrent server begins by waiting for the establishment of a TCP connection from a new client. Once the connection is established, the server forks a child process which then handles all the communication with that particular photo client. The parent server process returns to waiting to accept new photo clients.

The command line to start the server is simply:

./***server***

where

photonew**ij**.jpg indicates the name of the **j**th photo for the **i**th client in the server's photo gallery.

and

server_id.log indicates the file which records significant server events relative to client ***id***.
The forked child process of the server is responsible for communication to one photo client using the three emulated protocol layers.

The Photo Server Application/Network Layer

The ***server application layer's*** responsibility is to take **256 byte** photo chunks out of network packets to reconstruct the client's photos and write them out to the correct files in the photo gallery. The ***server application layer*** interrogates the end-of-photo indicator byte in the packet to know when the current packet is the last packet for a photo 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 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***, *receiving* an ACK packet from the ***server network layer*** and *sending* it as a data frame, and *sending* an ACK frame back to the client via 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.** When the client closes the connection to the server, the forked child process of the server terminates.

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.

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***. Note – the ***server data link layer*** needs to send an ACK frame when a duplicate frame is detected due to possibly damaged ACK frames. The server records significant events associated with client ***id*** that includes frame received, frame received in error, duplicate frame received, ACK frame sent, ACK packet sent and packet sent to the network layer in ***server_id.log***.

Frame Error Simulation

Since real 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** in every **11th** 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 11, 22, 33, … 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 simplicity in this assignment that all data frames 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*.**

Assignment Hints

- **[Debug]** Build and debug your programs in stages. Begin by getting the client and server working without processing errors and without a timer. Then add the error generating functions and the timer mechanisms to the client. Initially, the client and server can exist on the same machine if this simplifies debugging. However, at least one member of the programming team needs to focus on making sure the final project permits the client and server to exist and be tested by the TA on any **arbitrary** CCC Linux machine.

- **[Error Detection]** While **CRC** at the bit level will be discussed in class, it is recommended that you use a two-byte **XOR folding** algorithm of all the frame bytes to create your error-detection bytes. While this error detection scheme is not as strong as **CRC**, it is adequate to handle the single bit errors being induced by the emulated data link layer. For ACK frames, the error-detection bytes simply become a copy of the two-byte sequence number.

- 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, but given it is the last assignment at the end of the term, you may have to resort to polling if your team is unable to successfully implement **select**. It is important that your README file identify which technique your program employed.

- **[Performance Timing]** You **must** measure **the total execution time** of the complete emulated transfer of all the photos for each client. Be sure to print this result out in readable form in the file ***client_id.log***.

- **[Timers] The PAR protocol can fail if there is a premature timeout. Set the timeout period on your timer large enough to insure NO premature timeouts. However, setting the timer "crazy high" will cause your photo transfers to run quite long.**

- **port numbers:** You can "hardwire in" the well-known port number of the server for this assignment. Do not use low port numbers and we will establish a scheme for unique port numbers for each project team.

- The **actual content of the photos** written by the Photo Server should **exactly** match the photos 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 photo to test out the client prior to the TA releasing the official test photos.

What to turn in for Program 3

The TA will make available official test photo files a couple of days before the due date. Turn in your assignment using the Linux ***turnin*** program. Turn in the source files for the ***clients*** and the ***server***, a **README** file and a make file that the TA can run to test your client and server. Tar all your files together before submitting to turnin. **As this is the last assignment at the end of the term, if your program only works partially, to maximize your potential for partial credit for programs that do not fully work properly it is quite important that your README file identify clearly which parts of your program work and you also identify those parts of your program which still do not work when you turned in this program. If you need separate unique 'driver' programs to show those components which do in fact work, please provide them and explain how the TA should run them to give your team as much partial credit as possible.**