Programming Assignment 3 (60 points)
Due: 11:59 p.m. Tuesday, December 10, 2002

Concurrent Server Using Go Back N

This assignment builds on the experiences of Program 2 that implemented a simple client-server protocol on top of an emulated physical layer. This program is to be completed in two person teams or by individuals. At this stage of the course, the **only** single person projects that are encouraged are situations where your partner has dropped out of the course.

The assignment is to build a **concurrent server** that handles requests from two or more clients. **[*Unix socket calls are used with TCP as the physical layer for transmitting between clients and a concurrent server*].** Both the clients and the server will have a small application layer protocol that defines the interaction between client and server. Furthermore, the architecture includes a small network layer to deal with the conversion from messages-to-packets-to-frames. **Note:** Some of the data link layer issues from Program 2 can be simplified in Program 3.

The command line for the client program is: ***client script****num*

where

script*num* is an input parameter that indicates (indirectly) which client this is.

Hence the command ***client script1*** starts up client 1 and ***client script2*** starts up client2.

Application Layer

The **client application process** read its "scripted actions" from the file *script**i**.txt.* That is, client1 reads from ***script1.txt*** and client2 read from ***script2.txt***

Each **client application process** sends requests (one at a time) to the **server application process** of the form:

command number message

where *command* can be one of the following:

r	indicates read a message from the server
q	indicates quit the conversation and close the client connection
w	indicates write a message to the server

{**r**, **w** and **q** are ASCII characters}

number is an integer between **1** and **15** indicating the location of the message in the server database

message is a text message. The maximum size of a message is **320** bytes.

The **server application process** begins by reading into memory the original database of **15** messages from the input file ***serverbase.txt***. The server application process handles client application requests to read or write a message from/to the database.

When a new client connects to the concurrent server, the server will ***fork*** a child process to handle all interactions with that client. The server child process begins with a fresh copy of the original database. The server child process responds to the client's requests using the child's copy of the database.

When the client application issues a **read** request, the server child process sends a copy of the requested message back to the client.

Read Example

client sends message: **r 6**
server child extracts message **6** from its copy of the database and sends it back to the client.

When the client application issues a **write** request, the server child process ***overwrites*** the received message in the correct place in the child's copy of the database. The server must send back a **response message** to the client to indicate that the **write** request has been completed.

Write Example

client sends the message: **w 4 Duke Blue Devils – will they repeat as National Champs?**
server child overwrites message 4 in the child's database with the text:

Duke Blue Devils – will they repeat as National Champs?

Thus, each server child process maintains a ***separate copy*** of the database for the client it is serving.

Quit Example

When the server receives a **quit message** from the client, the server child process prints out the database to the appropriate *server**i.**log* file (where **i** is the number of the client). ***See the physical layer below for how the server knows which file to write out.*** The server child then sends a response to the client before terminating the child process. **Note:** You need a "clean" way to terminate the server.

The client application process sends a new command to the server application process after it receives a message back from the server or after it receives a response message from a write command.

Network Layer

The network layer receives the messages from the application layer and converts the message into packets. The maximum size of a packet for this assignment is **60** bytes. Packets are sent to the data link

layer to be converted to frames for transmission. Note – the network layer continues to send packets until blocked by the data link layer.

The network layer also receives packets from the data link layer. It reassembles the packets into a message to send to the application layer. **Note: the network layer will need to have a mechanism to determine the last packet in a message.**

Data Link Layer

The data link layer receives packets from the network layer, creates frames, and sends frames to the physical layer for transmission. The data link layer also receives transmitted frames from the physical layer, extracts the payload, reassembles packets, and forwards packets to the network layer. The maximum size for the frame payload is **32** bytes. You must design the "overhead" bytes of the frame to implement a **Go Back N sliding window** protocol. If it simplifies your task, framing bytes and byte stuffing are not necessary for assignment 3. **However, if it easier to keep these functions in this assignment that is fine.** As in program 2, your design needs to include an error-detection byte. Your design will need to include sequence numbers in the frames and a mechanism for handling ACKs**.** The minimum frame size is your choice, and it is your design decision whether to *piggyback* ACKs or send separate ACK frames. Due to the **request/response** nature of the application layer, ACK timers are not necessary for this assignment.

The goal of this assignment is to implement a **Go Back N sliding window** scheme at the data link layer. For **full credit** you must implement a sending window size of **four** frames or higher. The data link layer continues to receive packets from the network layer until its sliding window is full of unACKed frames. This requires **multiple timers** on both the client and server side. If you are short on time or run into problems, you should fall back to implementing a one-bit sliding window with a single timer on both sides of the connection.

Client Process Flow

Each client will call the physical layer to establish a connection to the concurrent server. The data link layer then gets packets from the network layer, puts together a frame and gives it to the physical layer to send. The data link layer flow will then depend on events coming from the network layer, the current availability within the sliding window, and events coming from the physical layer. The client process will terminate when the response message to the **quit** message is received at the client application layer.

Flow of Server Child Process

Each data link layer server child process waits for frames from the physical layer and passes packets up to the network layer. Similar to the client side, the flow of the server data link layer depends on whether there is traffic from the server to be sent back to the client (either frames with packets or ACK frames).

Each data link client records significant events in a log file *client**i**.log*. Each data link layer server child records significant events in a log file *server**i**.log*. Significant events include: packet sent, frame sent, frame resent, ACK sent, ACK resent, frame received successfully, frame received in error, ACK

received successfully, ACK received in error, duplicate frame, and timer expires. For logging purposes identify the packet and the sequence number of each frame for each event. Begin counting packets and frames at 1.

Physical Layer

The physical layer uses Unix sockets to send the constructed frames as actual TCP messages between the clients and the concurrent server. When the client physical layer first establishes a connection to the concurrent server, it **must** send one TCP message to the server child process to identify itself (i.e., the client with input parameter ***script1*** sends a message containing ***client1*** to the server child process). This tells the server child where to print out the final database.

Simulating Errors

Force every 6th frame sent by each client to be in error by flipping any single bit in the error-detection byte prior to transmission. Force every 7th frame sent by each server child to be in error using the same flipping mechanism. (i.e., frames 6, 12, 18, … sent by each client will have a transmission error and frames 7, 14, 21, … sent by each server child will be in error.) Each frame with a forced error should be resent correctly on the second try.

Hints

- **[Design]** Plan your design in a modular fashion such that if everything is not totally working, you can turn in an output that shows **exactly** what is working. Relaxing the sliding window scheme is one option.

- **[Documentation]** Your commented program **must** have a special section to explain the details of your specific design decisions. **Remember: This a team project and all routines must include specify the author as part of the documentation!! Team members may not receive the same grade on an assignment due to uneven workload.**

- **[DEBUG]** Include print statements in the various layers while debugging. You should consider some type of verbose debugging flag that can be turned on and off.

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

- **port numbers** - Your clients should have unique port numbers and the clients should treat the server port number like a "well-known" port number. ***{Conceivably, you could register your database server with the oracle from program 1 and the client processes could get* IP *address and port number from the oracle.}***

What to turn in for Assignment 3

The three official tests file (*script1.txt, script2.txt, serverbase.txt*) will be made available a few days before the due date. Use *turnin* to turn in the two source programs *client.c* and *server.c*, and the client and server output files corresponding to running the programs using the TA's test files. The README file **must indicate to the TA the working and the non-working parts to receive partial-credit for program 3.**