CS 422 Fall 2014

Lab 4: Network File Servers and Symmetric Network Chat App

Due: 11/02/2014 (Sun), 11:59 PM

Objective

The objective of this lab is to implement a TCP-based concurrent network file server and design a network chat service.

Problems [280 pts]

Problem 1 (120 pts)

Now that we have a basic understanding of how to achieve reliable transmission using TCP which implements sliding window ARQ, we will use TCP to build a network file server. The server process, fsserver, takes as command-line argument

% fsserver server-port

the port number on which it listens for client requests. The file server follows a concurrent server design where the server process parses a client request but hands off the actual servicing of a client request, i.e., transmitting a file to a client, to a worker process. A client request is a string that specifies a full pathname of a regular file. In our case, the regular file can be on any file system (FS) that is mounted by the sslab machines in LWSN B158 that fsserver has access to. fsserver, upon receiving a client request, uses access() to check if the requested full pathname exists and is readable by the server process. If a requested file does

not exist or is not readable (the submitted string may even be ill-formed, for example, containing a backslash), fsserver responds with a brief message "INVALID REQUEST", closes the connection (socket descriptor returned by accept()) and returns to waiting for the next client request. Thus when access() fails, the file server acts as an iterative server by responding to the client itself which is efficient since the response message is short and fixed.

If fsserver determines that the full pathname exists and is readable, then it forks a child process and let's the child process handle the actual transmission of the requestsd file to the client. The server process returns to waiting for the next client request. Recall from system programming what resources (e.g., file descriptors) are shared between parent/child processes so that the hand-off from parent to child can be made as efficient as possible. The child (or worker) process uses the write() system call to transmit the requested file, and upon completion, closes the connection and terminates. Note from Problem 3, Lab 2, that it is good programming practice for parent processes to invoke wait() when a child process terminates. Reuse the SIGCHLD signal handler to perform this task.

When the child process transmits the content of the requested file to the client, it must first read a block of bytes using read() that interfaces with the file system, then use the write() system call on the socket descriptor to transmit the block of bytes to the client using TCP. Thus read()/write() forms a loop that terminates when all the bytes of the requested file have been read and transmitted. More importantly, from a performance perspective, you will need to decide how many bytes to read during each read() system call to the Linux kernel. Through your knowledge of operating systems and experiments that confirm your understanding of Linux file systems, determine what is good block size to use when calling read(). Use the same block size when performing write() (note that disk I/O is the principle bottleneck in a disk-based FS).

Discuss your selection of a suitable block size that leads to good file server performance in lab4writeup1.pdf and put it in mylab4/p1. Use gettimeofday() to obtain objective measurements that support your selection of block size. For example, you can insert gettimeofday() before the first call to read() and after the last call to read() to get a wall clock time estimate of the disk I/O (and network I/O) overhead. Test your file server on larger file sizes to gauge performance. The client fsclient takes command-line arguments

% fsclient server-IP server-port pathname filename

where the first two arguments specify the server's IP address and port number, the third argument specifies the full pathname of a file to be downloaded from the file server, and the fourth argument specifies the filename (in the current directory of the fsclient process) into which the downloaded file will be saved. Verify correctness of your file server app, that is, that the downloaded file saved as filename is the same as pathname. Submit the file server client/server app code in mylab4/p1.

Problem 2 (140 pts)

Not long after the Stone Age but before the full advent of the X Window System and GUI-based applications, a popular app for performing chatting/instant messaging was talk. talk was a symmetric client/server app, or what is also referred to as a peer-to-peer (P2P) app, where there is no separation between server and client. That is, the app talk is both server and client. This is different from the asymmetric client/server apps we have encountered in the labs to date.

You will develop a UDP-based talk app, call it mytalk, where mytalk is invoked with a single command-line argument

% mytalk my-port-number

which specifies the port number that mytalk will use to communicate using UDP. When mytalk runs, it will bind itself to my-port-number and print a prompt "> " to stdout and wait for stdin input from the user. A user commences a chat by typing the IP address and port number of the peer (i.e., chat party) that he/she wishes to converse with. mytalk sends a UDP packet to initiate a chat session by inscribing the string "letschat" in the datagram's (UDP packets have historically been referred to as datagrams) payload. If the peer is not responding within 5 seconds (use SIGALRM handler as in Lab 2 to affect timeout) mytalk outputs to stdout a suitable message that indicates no response, prints the "> " prompt and waits for user input. The user may initiate another connection or quit the app by typing 'q' (followed by RETURN/newline). The peer may not respond because mytalk is not running at the specified IP address/port number, or the UDP control message "letschat" was lost.

If mytalk is running at the peer (it's the same app, i.e., binary hence no distinction between server and client) it will print to stdout the message "chat request from peer-IP peer-port" where peer-IP and peer-port are the

IP address and port number of the chat initiating peer. mytalk prints the "> " prompts and waits for user input. If the user types 'c' then a chat connection is completed by sending a UDP datagram with payload "ok" to the requesting party. If the user enters 'n' then the payload "ko" is sent indicating that the peer does not wish to chat. mytalk at the initiating party prints "doesn't want to chat" and returns to waiting on "> ".

Once a chat connection has been established, mytalk must juggle two tasks. First, it must read the characters (i.e., message) typed by its user and when RETURN/newline is input send a UDP packet with the message as its payload. The message in the payload is preceded by the ASCII character 'M' indicating that what follows is a message. If the user enters the single character 'd' (followed by RETURN/newline) then this will mean that the user is done chatting and mytalk transmits a UDP packet the single character 'D' as payload indicating disconnect. mytalk at the other end prints "chat terminated" and returns to the "> " prompt.

Second, while a user is in the midst of typing a chat message, a chat message from the peer may arrive that needs to be displayed. The Linux kernel will raise the SIGPOLL (equivalently SIGIO) signal when a UDP packet arrives. The code structure of mytalk should be such that after a chat connection has been established, its main body makes a blocking call to read() to read user input from stdin one byte at a time until newline. When newline is entered, mytalk transmits the entered message in a UDP datagram as specified above. Printing of incoming chat messages from the peer are handled asynchronously by registering with the Linux kernel a SIGPOLL handler that handles the task of printing the received chat message to stdout. Note that there are subtleties that must be carefully handled such as two chat messages arriving from the peer back-to-back. Since the Linux signal handling system does not keep track of how many UDP packets have been received, before your SIGPOLL handler returns it needs to check that there are no further chat messages waiting in the kernel buffer.

To focus on the networking component of the app design, we will not utilize graphical user interfaces. talk used the curses package which allowed the a character terminal to be split into two halves so that a user could type messages in one half and see the peer's chat messages in the other. We will not be using curses which is a bug prone scary proposition (compared to today's GUI programming environments curses programming is like sleeping on a bed of nails). Therefore will manage a simple hack to not overly confuse the mytalk user and make the chat sessions legible. For example, if a user is in the midst of typing "Let's go to dinn" and the message "How about dinner?" arrives, we don't want the SIGPOLL handler to just dump the received chat message to stdout which the user would see as "Let's go to dinnHow about dinner?" Instead, to

make the chat session more legible, the SIGPOLL handler will print on a new line the received message ": How about dinner?" preceded by ": " as a tag/marker. It will also print the partly typed message "Let's go to dinn" on a new line so that the user can continue typing the message where he/she left off. From a programming technique perspective, note that you are juggling multiple signals (SIGALRM, SIGPOLL) and the socket descriptor must be set in asynchronous mode using O_ASYNC and O_NONBLOCK in fcntl(). The latter's use and syntax varies across UNIX/Linux versions and should be separately tested to confirm correct functioning. This is part of UNIX/Linux network programming that must be carefully handled when porting across different platforms. Implement and test your mytalk app on a pair of lab machines and submit the code in mylab4/p2.

Bonus Problem (30 pts)

Providing adequate comments within code is very important but sometimes not emphasized enough when focus is on technical content. In Lab 4, we will provide incentive to earn bonus points by doing a good job commenting code. Points will be deducted if comments are deemed inadequate which applies to all labs.

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Create a directory named mylab4 under which the subdirectories p1 and p2 and code submissions are organized. Go to the parent directory of mylab4 and type the command

turnin -c cs422 -p lab4 mylab4

Please note the assignment submission policy specified in the course home page.

Back to the CS 422 web page