CS 422 Fall 2014

Lab 2: Basic Asynchronous and Concurrent Network Server Design

Due: 10/05/2014 (Sun), 11:59 PM

Objective

The objective of this lab is to practice asynchronous and multithreaded programming needed in the context of full-fledged concurrent server network applications.

Reading

Read chapters 5, 6 and 11 from Comer (textbook).

Problems [270 pts]

Problem 1 (70 pts)

This is an extension of Problem 1 in Lab 1. Modify the myreminder app, call it mobilereminderd, such that it sends reminder notifications to a mobile user. The client app, mobilereminder, runs as a process on some IP enabled device with IP address X and port number R. The mobile client accepts three command-line arguments

% mobilereminder server-IP-address server-port-number secret-key

where the first two arguments specify the IP address and port number of the server process mobilereminderd, respectively; secret-key is a alphanumeric ASCII character string. The server is run with two additional command-line arguments

% myreminder reminder-file server-port-number secret-key

where server-port-number is the port number it should bind to and secret-key specifies a secret key to be used by a roaming client when contacting the server. The client process mobilereminder uses UDP, as in Problem 3 of Lab 1, to send a datagram (i.e., UDP packet) to the server where the payload contains the secret key. This acts as a registration message. The server process, mobilereminderd, upon receiving a UDP packet with alphanumeric payload (those containing non-alphanumeric values should be outright rejected/ignored), checks if the secret key matches the one provided as its second command-line argument. If they do not match, the registration message is ignored. If they match, the client's IP address X and port number R inscribed in the UDP header are remembered and all future alarms that trigger the SIGALRM handler cause it to send a UDP packet to the registered client with the reminder message (in reminder-file) as payload.

When a client roams, its IP address and port number may change. In our set-up, instead of a mobile device roaming with potentially changing IP addresses and port numbers, we will have the user roam between the Linux PCs in the LWSN B158 lab. For example, you might first login to sslab01, run the server mobilereminderd, then login to sslab02 to run mobilereminder. After verifying that you receive reminder messages that are printed on stdout, you quit the client app, login to sslab03 and run mobilereminder again. When the server receives successive valid registration messages (i.e., the secret keys match), it updates the client IP address and port number based on the most recent datagram received.

Write code that is modular with each function stored in its own .c file. Use Makefile to automate compilation. Your code must be adequately documented. Put your files in a subdirectory mylab2/p1. Additional submission instructions are specified under turn-in instructions below.

Problem 2 (100 pts)

A serious drawback of the mobile reminder app in Problem 1 is that UDP packets may get lost in which case reminder messages may not make it to the client or the server is unaware of updated client registrations. Without losing the strength of the UDP based app, its lightweight nature, add your custom reliability mechanism on top of UDP (you may not use TCP) that aims to achieve reliable client/server message delivery. For example, a "solution" might be as simple as: client sends a registration packet, server immediately replies with an acknowledgment packet. If a server acknowledgment (ACK) packet does not arrive within 5 seconds, the client retransmits the registration message to the server until an ACK arrives. A SIGALRM timer mechanism, similar to that described in the Bonus Problem of Lab 1, may be used for this purpose. The same method may be employed to assure that server responses are received by the client.

One further complication that management/control packets introduce is that they need to be distinguished from packets carrying payload (i.e., reminder messages). Devise your own scheme that accomplishes this and provide a description it in a separate pdf file, lab2writeup2.pdf. Deposit lab2writeup2.pdf in mylab2/p2 along with your code. If you choose to implement a different UDP-based solution, please do so and describe its working in lab2writeup2.pdf. Test that your implementation works correctly by having sendto() (both at client mobilereminder and server mobilereminderd) not transmit every other message. Do this by writing a wrapper function, mysendto(), which takes the same arguments as sendto() but adds a conditional check at the start to return without calling sendto() every other function call.

Problem 3 (100 pts)

The ping-like application developed in Problem 3 of Lab 1 follows an iterative server design: the server, after receiving a ping UDP packet from a client, performs the requested task itself. In our case, sending back a UDP ping response packet. In a concurrent server, the server process spawns a child worker process and delegates execution of the task to the worker process. In further optimizations, a number of worker processes may be pre-spawned to reduce time delays introduced by process creation overhead.

As an extension of the ping-like app, myping, rewrite the app so that it follows a concurrent server design. The server process mypingd still waits on port 50000 for UDP ping requests. When a packet arrives, it creates a worker process by calling fork(). Recall that fork() "returns twice": in the parent process, it returns the child process's process ID (PID). In the child process, it returns 0 which helps the child process recognize itself as a child. The parent process's code after fork() branches back to waiting for the next UDP ping packet from a client by calling recvfrom(), thus forming an infinite loop. The child code following fork() uses sendto() to transmit a ping respond to the client. From a programming perspective, noting what parent/child processes share and how to make the child process seemlessly access data structures that identify the client process (IP address, port number, payload received) becomes key. Also, in the revised software design, only the server code needs to be changed with the client code remaining as is (i.e., backward compatible).

One further component that needs to be addressed in a concurrent server design is management of child processes that terminate before their parent. In the concurrent server version of mypingd, a child process terminates after it transmits a UDP ping response datagram to the client. If a parent process does not execute waitpid() or wait() after a child process terminates, it becomes a zombie process that occupies space in a kernel's process table. A common programming practice to "provide a proper burial" of zombie processes is to register a SIGCHLD signal handler in the parent process that is invoked when SIGCHLD is raised (upon child process termination). Add a SIGCHLD handler to the server code (always use sigaction()) that cleans up zombie processes when child processes terminate.

Write code that is modular with each function stored in its own .c file. Use Makefile to automate compilation. Your code must be adequately documented. At least 15% of the points received will be deducted if comments are deemed inadequate. Put your files in a subdirectory mylab2/p3. Additional submission instructions are specified under turn-in instructions below.

Bonus Problem (30 pts)

As an extension of Problem 3, consider a concurrent server design where a pool of worker processes are pre-spawned to mitigate process creation overhead that can contribute to delayed response to client requests. How would you go about implementing inter-process communication (IPC) between parent and child worker processes so that task delegation to a child process incurs least overhead? Discuss your reasoning in a write-up lab2writeup4.pdf and submit it in mylab2/p4. *Note: The Bonus Problem is optional. That is, bonus points count toward the 50% that lab assignments contribute to the course grade.*

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Create a directory named mylab2 under which the subdirectories p1, p2, p3 (and p4) and code submissions are organized. Go to the parent directory of mylab2 and type the command

```
turnin -c cs422 -p lab2 mylab2
```

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

Back to the CS 422 web page