CS333 Spring 2016

Lab 2 (Take Home)

Goal

In this lab, we will build a file server that serves files from its disk to clients over the network. We will also build a multi-threaded client to simulate multiple concurrent users trying to download files from the server, and test the server performance with a large number of concurrent users. The goal of this lab is to get you comfortable with writing basic multi-process / multi-threaded applications, and understanding their performance. We will build upon this simple file server in subsequent take-home labs.

You must program this, and all subsequent labs in C/C++. Of course, you may use a scripting language to run experiments, analyze results, plot graphs, etc. But the main code must be in C/C++.

Before you start

- At a very basic level, the client and server are socket programs that communicate over TCP network sockets. You must revise basic socket programming from your networking course. You will find several tutorials online. Simple client and server socket programs are provided for your reference.

- The simple server socket program can only handle one client at a time, because the `accept` and `read` system calls are blocking. For example, when the server executes `read` on the socket of one client, the program blocks until data arrives from that client. Any new clients that try to connect to the server will be unable to do so. Therefore, a good way to build servers capable of handling multiple clients at a time is to spawn a new process (or thread, but we will go with processes for now) for every new client connection at the server. The new process will service the single client it was created for, blocking for the client reads, and terminate when it has finished serving the client. Meanwhile, the original server process will continue to wait for and accept new connections. Your server must do the same. Again, you will find several helpful webpages online on how to build multi-process servers that can handle multiple clients concurrently.

- Your client will simulate multiple users by using multiple threads in the client program. You will write C code with the POSIX threads (`pthreads`) library. Learn about how to create, execute, and terminate threads using `pthreads`.

- The most important part of the preparation for this lab is to identify two separate machines to run the client and server programs. There are several options: you may use two virtual machines (VMs) on the same physical machine (say, your laptop), you can use two separate machines (laptops or the lab desktops) connected over the CSE LAN, or you can connect the two machines via

your own switch and Ethernet cable setup. You are free to choose any setup of your choice, and the answers to the various exercises in the lab will depend on your setup. Let's understand how.

Ideally, we would like the clients to be able to download files from the server as fast as the server disk can retrieve files. However, you may not able to get this maximum performance from the server always, depending on what your setup is. Typical disk bandwidths are a few tens of MB/s (i.e., a few hundred Mbps), so your server (if coded well) should be able to achieve this throughput. However, network speeds may be lower than this, because while most machines have 1Gbps Ethernet cards, the shared network and switches may not support 1Gbps throughput. So, the throughput the clients see might be limited not by the server's disk but by the network. (Note that when you run your client and server on two VMs on the same physical machine, no packets go out into the external network, you get a very high bandwidth between your client and server, and this point above does not apply.) Finally, if your client machine is not as powerful, you may not even generate enough requests to load the network or the server. In summary, the performance bottleneck in your setup very much depends on the relative speeds of the client and server machines and the network in between.

You can use any setup of your choice to solve this lab, as long as you can explain the experimental results successfully in the context of your setup. You may revisit your setup and fine tune it as you go along solving this lab. However, please use the same setup to answer all the exercises, so that your answers are consistent. You can use a tool like `iperf` to measure the maximum network bandwidth between two hosts. A tool like `iostat` will give you the disk read bandwidth when running a disk-bound program (see `disk.c` from Lab 1).

Building the client and server

Client-server protocol. The file server runs on a known IP address and port. The clients connect to the server over TCP and issue a command of the form `get <filename>` over the TCP connection. The server reads the command from the socket, opens the requested file on its disk, reads data from the file, and writes it back into the socket. The file may require multiple rounds of reading from disk and writing to the socket. Once the entire file has been sent to the client, the client and server close their sockets. The client will not send any acknowledgement of the file to the server. Any error in any intermediate step (e.g., requested file not found) will also result in the client or server terminating the connection. A client can download only one file over one TCP connection, and must reopen a new connection for another file.

For all labs involving the file server, we assume that the working directory of the server process has a folder called `files`, which has 10,000 files of size 2MB (`foo0.txt`, ..., `foo9999.txt`). You would have created this corpus of files during one of the exercises in Lab 1. You may assume that the client requests only one of these files. For example, the commands sent from client to the server would look like ``get files/foo42.txt".

You may assume that all request strings sent by the clients are well-formed, and that all requested files exist on disk.

The file server. You must write a program `server-mp.c`, which takes a port number as its commandline argument. Here is how the server must be run.

```
$./server-mp 5000
```

The server program opens a TCP socket on the given port, to listen for incoming client connections. For every new client connection that is established via the `accept` system call, the server spawns a new

process, hands over the handling of the new client socket file descriptor to the new process, and goes back to listening for incoming connections in its original (parent) process. The new server child process must read the client request from the socket, fetch the file from the disk, and serve it to the client. The child process closes the TCP connection and exits after completing the file transfer. The parent server process must periodically reap its dead children.

Note that file reads from the disk must be done in multiples of disk sector size, which is usually 512 bytes. You can read files from the disk and write to socket in block sizes of 1024 bytes or 2048 bytes or any such reasonable value.

The client program You will write client program `multi-client.c`, that will emulate multiple users downloading files simulatenously from the file server. The client program spawns a certain specified number of threads at the start. Each thread in the client program does the following: it creates a new socket, connects to the server, requests a file from the server, and reads from the socket for as long as the server is sending data. The file downloaded from the server is not written to the disk at the client; the data is simply read from the socket and discarded. After the server finishes sending the entire file, the server process closes the socket from its end, and a `read` at the client doesn't return any data anymore. At this point, the client thread closes the socket, sleeps for a certain duration or "think time", and starts another round (from creating a socket) all over again. Each client thread continues to request files from the server in this manner until a certain specified duration of time elapses. (The clock can be checked after every round to see if time is up; there is no need to stop midway during a file download.) Once the experiment duration is up, all threads terminate, the parent `multi-client` process joins all threads, and exits.

Now, which file should each emulated user request from the server? Your client program can run in two modes. In the "random" mode, each emulated user requests a random file from the 10,000 files in the server's corpus. In the "fixed" mode, all emulated users always request only one fixed file (say, `foo0.txt`) from the server.

The client program takes the following as commandline arguments: the server IP address, server port, the number of concurrent users to emulate (which should be equal to the number of threads spawned by the client process), the duration of the experiment for which the threads should run, the think time between requests, and the mode (random or fixed).

One of the goals of this lab is to measure the performance of the file server. We will quantify the performance of the server by two metrics: *throughput*, defined as the number of requests the server can satisfy per second, and *response time*, defined as the time taken by the server to serve a file to a client. These values will be measured by the client program. At the end of the experiment, `multi-client` must print the following statistics: the average throughput in requests/sec (which is obtained by dividing the total successful requests completed by all threads divided by the time of the experiment), and average response time of the downloads (measured as the time from sending the get request to the file download completing, averaged over all successful downloads across all threads). To be able to print out these statistics, each thread must update the response time and count values after completing each round, and `multi-client` must aggregate statistics across all threads at the end of the experiment. Exercise caution is updating counts and other statistics across all threads: if you have a common variable, then the threads will need to lock to access that shared variable. A better solution would be to have an array with as many entries as the number of threads, and make each thread update only its slice of the array. In other words, make sure that no two threads update the same counter/variable, to ensure that the statistics are updated currently.

Here is a sample run of the client. This `multi-client` program is run to emulate 10 concurrent users, each of which requests random files from the server for 300 seconds, sleeping for 1 second between

downloads. Also note how the statistics are printed out at the end of the program (your format may differ).

```
$./multi-client 10.129.5.195 5000 10 300 1 random
Done!
throughput = 36.127 req/s
average response time = 0.123 sec
```

Tips and guidelines

Below is a suggested step-wise procedure for you to incrementally build the client and server, and some helpful tips.

- Start with building a simple client that does one download, and a simple server that will serve this download. This basic step will help you iron out the basics of socket programming, reading files from the disk, and so on. Check that the single file is downloaded correctly, by printing out its contents for example. You may want to keep a copy of this simple client and server for debugging in future labs as well.
- Next, make the server a multi-process application, by forking off a new process for every new client. You can test the correctness of this server by running two of your single clients in two different terminals, connecting to the same server simultaneously. At this stage, you should carefully work out a plan for how the server plans to reap its dead children (who would have terminated themselves after serving a file to the client they were created to serve). Look up the `wait` system call, especially its `waitpid` variant.
- Next, convert your client into a multi-thread application. Each thread must run for the duration of the experiment and terminate. The main client process must wait for all the threads to join at the end, print statistics, and finish. Make sure that all threads access and update different variables (say, by assigning a slice of an array for each thread), so that multiple clients do not access the same variable without proper locking.
- As with any C code, be careful when you dynamically allocate memory. Make sure you free the memory before the end of the program, in order to avoid memory leaks.
- Avoid using NFS to store files that your server must access etc. NFS access has worse performance than running code and accessing your local disk. So be mindful of where your server files are hosted.

Exercises

Once you build your client and server, run some basic tests for correctness. Your programs should run correctly without any crashes for several tens of simultaneous users, and for several hundreds of seconds. The processes should not leave behind dead children or leak memory. The statistics printed out at the client should all make sense. Once you have some basic confidence in the correctness of your code, let's proceed to do some exercises that test the performance of your server under different scenarios.

Note: Whenever you read a set of files from the server, a copy of some of the files will exist in the disk buffer cache in main memory of the server. Subsequent accesses of the file will not go to the disk. This caching may lead you to get different numbers for different runs of the experiment. To avoid confusion, we suggest that you always clear your disk buffer cache at the server, before you start a new experiment. The following commands (to be run as superuser) will flush the disk buffer cache.

```
# sync
# echo 3 > /proc/sys/vm/drop_caches
```

Carefully record your observations from the exercises below. **Note: the answers to your exercises below will depend on the setup you have used to connect your client and server machines; please interpret your results carefully in the context of your experimental setup. There is no one right answer that everyone should get, but all your answers should be consistent and must make sense.**

1. Describe the setup and hardware specifications of your client and server machines, and how they are connected. Next, measure two things to help us understand your setup better: (i) Measure the maximum read bandwidth of the disk on the server in MB/s (megabytes per second). That is, if the server were to be reading files continuously from the disk, how many MB/s can it read? Approximately how many requests/sec does this translate into, given your download file size? There are several ways to measure the maximum disk read bandwidth. For example, run a disk-intensive benchmark (like `disk` from Lab1), and a tool like `iostat` can show you the rate at which the disk is reading data. (ii) Measure the maximum network bandwidth you can get between your client and server machines using `iperf`. Justify the network bandwidth you observed in the context of your experimental setup. Again, calculate how many requests/sec this bandwidth translates into, by multiplying/dividing with the file size suitably.

2. Run experiments with your `server-mp` and `multi-client` by varying the number of threads in the client N (suggested values are 1, 2, 5, 10; but you may use your judgement to pick other values also, based on your setup). The other parameters at the clients should be set as follows: random reads, no sleep time, experiment duration of 120 seconds. As you increase the load on the server by increasing N, observe what happens to the throughput and response time reported at the client. Now answer the following questions:

   (a) What is the optimal value of N (among those you have experimented with) that *saturates* the server? In other words, what is the lowest value of N that loads the server fully to obtain the highest possible throughput (in terms of requests served per second)?

   (b) What happens to the throughput and response time of the server for values of N lower or higher than this optimal value? You may show graphs, or explain your observations in text.

   (c) When the server is operating at saturation, what was its bottleneck resource? Explain how you identified the bottleneck.

   (d) What is the server throughput at saturation (at the optimal value of N you discovered above)? Justify this number in the context of the bottleneck resource identified. (For example, if you identified the network as the bottleneck, then the server throughput at saturation should approximately match the maximum network bandwidth as measured in exercise 1.)

3. Repeat exercise 2 above with a sleep time of 1 second between user requests (everything else remains the same). Because each user sleeps between file downloads, the client program will not

stress the server as much as before for any given value of N. So you may have to increase the values of N you explore to much higher values (suggested values are 1, 5, 10, 15, 20, 25, etc.) in order to saturate the server.

4. Repeat exercise 2 above, but with the client mode changed to "fixed" from "random".

Submission and Grading

You may solve this assignment in groups of one or two students. You must submit a tar gzipped file, whose filename is a string of the roll numbers of your group members separated by an underscore. For example, `rollnumber1_rollnumber2.tgz`. The tar file should contain the following:

- `report.pdf`, which contains your answers to the exercises above. Be clear and concise in your writing.
- Your code `server-mp.c` and `multi-client.c`. Please make sure your code is well documented and readable.
- An optional makefile to build your code.
- `readme.txt` describing your experimental setup, and instructions on how to run your code.

Evaluation of this lab will be as follows.

- We will run your code and check that it compiles and runs correctly without crashes for long periods of time.
- We will run our client with your server, and vice versa, to test that your implementation conforms to the protocol specified correctly. We will also check that the file is being correctly downloaded by printing out the contents for a small number of downloads.
- We will check that your code has no memory leaks (by running for a long time and observing its memory usage). We will also check that dead processes are being reaped correctly at the server.
- We will run your client and server code in a setup similar to what you have used, to reproduce some of your performance results.
- We will read through your answers to the exercises to make sure they are consistent with your code and experimental setup.