ECE 420 Parallel and Distributed Programming Lab 2: A Multithreaded Server to Handle Concurrent Read and Write Requests

Winter 2017

In this lab, you will implement a multithreaded server that can simultaneously handle multiple client requests, each request being a read or write operation applied onto a random position in a global array of strings stored on the server's main memory. To simulate multiple simultaneous requests, you will also implement a multithreaded client program that can launch read/write requests in Pthreads. The client and server should talk to each other via TCP sockets. You will be asked to optimize the processing speed of your server program and reduce the read/write latencies at the client threads based on what you have learned in this course.

1 Background

Most clients and servers communicate by sending streams of bytes over TCP connections, which make sure the bytes will be received error-free and in the order they were sent. A *socket* is an endpoint of a connection between a client process and a server process.

A server or (a service) can be uniquely identified by an `IP:port` pair, where a port is a unique communication end point on a host, named by a 16-bit integer, and associated with a process. For example, a server process with a socket address of 127.0.0.1:3000 runs on the localhost and on port 3000. A client can establish connection to a server as long as it knows the server's socket address, i.e., the `IP:port` pair of the server.

Sockets define operations for creating connections, attaching to the network, sending/receiving data, and closing connections in client-server communcation. Please see the provided code `server.c` and `client.c` in the development kit, which illustrate the communication between a simple echo server and a client. Specifically, the client process gets a text string from the user input and sends it to the echo server at 127.0.0.1:3000, which simply sends the received text string back to the client for display. Note that the server is a multithreaded server and can handle multiple client connections simultaneously. In other words, the provided code in `server.c` and `client.c` illustrates all the server-client communication functions you need for this lab.

1.1 The Case of a Single Client

We first explain what happens at both the server and client sides when there is a single client. The client does the following things:

Algorithm 1 Client actions in the sample code `client.c`

```
Create a socket using Socket(), which is identified by clientFileDescriptor;
Connect to the server 127.0.0.1:3000, which is another process running on the same host, using Connect();
Send data to the server using Write();
Receive the echoed data from the server using Read() and display it.
```

The server does the following things:

Algorithm 2 Server actions in `server.c` when there is a single client.

```
Create a socket using Socket(), which is identified by serverFileDescriptor;
Bind the server socket to the address 127.0.0.1:3000;
Listen on the server socket;
Block at the Accept() function until there is an incoming client connection.
Receive data from the client;
Send the the received string data back to the client.
```

1.2 A Multithreaded Server to Handle Multiple Clients

You may have already noticed that the server in `server.c` has employed multithreading to handle multiple simultaneous client requests. In this case, upon accepting an incoming client connection, the Accept() function will create a new socket (on a different port of the server) identified by the integer `clientFileDescriptor` for the accepted connection, and launches a new thread to handle the connected client. In the meantime, the server still listens to and accepts incoming clients on the *original* socket `serverFileDescriptor`.

For more background on stream socket communication, please refer to the following materials:

- Slides by Jeff Chase, Duke University (https://users.cs.duke.edu/~chase/cps196/slides/sockets.pdf)
- Beej's Guide to Network Programming (http://beej.us/guide/bgnet/output/html/multipage/index.html)

However, you don't need a deep understanding of network programming. For the purpose of completing this lab, it will be sufficient if you can understand and use the sample code provided in `server.c` and `client.c`.

2 Concurrent Writes and Reads to a Same Data Structure

In this lab, you will launch multiple threads in your client program to connect to a multi-threaded server. The server maintains an array of strings in its memory, and each client thread will perform either a write or a read to a random position (string) in the array. In the case of a read, the server sends back the corresponding requested string to the client thread, which will close the connection and terminate after receiving the returned string. In the case of a write, the server will update the corresponding string in the array with the new text string supplied by the client and return the updated string to the client. The client thread closes the connection and terminates upon receiving the returned string. Only 5% of all the requests will be writes and 95% will be reads.

Note that there could be multiple concurrent read or write requests operating on a same string in the array. In this case, we need to protect the critical sections and avoid race conditions. But how can we achieve the fastest processing speed? Think about the following questions. Should we use a single mutex for the entire string array or a different mutex for each string? What's the tradeoff here? Is there a better solution than the above two? Will read/write locks help? And should we use a read/write lock for the entire array or one for each string? What other optimization can you do to further enhance the performance?

3 Tasks and Requirements

Tasks: Implement a multithreaded server and a multithreaded client that performs read or write operations to a string array on the server.

1. Implement a multithreaded server that can handle multiple simultaneous incoming TCP connections from client threads. The server must communicate with clients via stream sockets, as shown in the sample code.

2. Maintain an array of n strings, `char** theArray`, in the memory of the server, where each string i ($i = 0, 1, 2, \ldots, n-1$) is filled with the initial value "String i: the initial value".

3. Implement a client program that can launch x simultaneous write/read requests to the server. Each request is launched in a separate Pthread and performs either a read or a write operation to a *random* string in `theArray` on the server. For a read request, the server returns the requested string to the user. For a write request, the requesting thread will update the corresponding string in `theArray` on the server to the following: "String *pos* has been modified by a write request", where *pos* is the id of the string that is updated. Each request is a write operation with a probability of 5% and is a read operation with a probability of 95%.

4. For a read request, the server will send the requested string back to the requesting client thread. Upon receiving the string, the client thread will display it, close the connection and terminate. For a write request, the

server will send the updated string back to the client. Upon receiving the string, the client thread will display it, close the connection and terminate.

Specific Programming Requirements and Remarks:

1. Check the sample code in "Development Kit Lab 2" for 1) how to perform client-server communication using stream sockets; 2) how to implement a multi-threaded server; 3) how to handle race conditions and critical sections with a basic approaching, i.e., protecting the entire array with a single mutex; 4) how to generate random integers in a thread-safe way using seeds. Refer to the *ReadMe* file for details on how to use them.

2. Your client should launch $x = 1000$ request threads.

3. For each client thread, you need to generate random numbers to determine the array position the thread is operating on, and to determine whether the operation is a read or a write. For a particular client thread i ($i = 0, 1, 2, \ldots, x-1$), the seed value used to feed its random number generators must equal to i, and the first random number generated in this thread must be the array position that the thread will operate on. This will help TAs to verify the correctness of your programs. Follow the style in `arrayRW.c` for random number generation.

4. Make sure your programs are correct and lead to correct read/write results.

5. Both your server and client programs should have the following arguments ONLY: 1) the port number on which the server is running; 2) the number of strings in `theArray`, namely n.

6. Measure the total time it takes to process all $x = 1000$ client requests that were launched. Rerun your client program for 100 times and repeat such time measurement for each run. Plot the CDF of your time measurements (e.g., by using `cdfplot()` in MATLAB). You may use `test.sh` in the development kit to repeatedly run your client program for a number of times. **Note:** for time measurement purposes, you may disable message printing to reduce overhead.

7. Optimize the request processing speed of your server program as much as possible.

Lab Report Requirements:

1. Describe your implementation clearly.
2. Discuss how you verified the correctness of your programs.
3. Explain the performance of your implementation(s) as the number of strings in `theArray`, namely n, takes the values 10, 100, 1000, and 10,000. Compare the performance of your server implementations, if you have implemented your server in more than one way, as n takes the values 10, 100, 1000, and 10,000.
4. For each of your server implementation(s), plot the CDF of processing time measurements collected from the 100 runs. Compare the mean or median processing times of different server implementations.
5. Explain your observations.
6. Please refer to the "Lab Report Guide" for the general requirements on formatting and content.

Submission Instructions:

Each team is required to submit BOTH a hard copy of printed lab report to the assignment box AND the source code on eClass. The report should be submitted in the assignment box on the 2nd floor of ECERF. The code should be submitted on eClass.

For code submission, each team is required to submit a zip file to eClass. The zip file should be named "StudentID-Hx.zip", where "StudentID" is the Student ID of **one** of your group members (doesn't matter which member) and "Hx" is the section (H1 or H2).

The zip file should contain the following files:

1. "readme": a text file containing instructions on how to compile your source files;

2. “members”: a text file listing the student IDs of ALL group members, with each student ID occupying one line;
3. “Makefile”: the makefile to generate the executable. Please ensure that your Makefile is located in the root folder of that zip file and the default “$make” command will generate the final executable programs;
4. All the necessary source files to build the executables.

Note: you must use the file names suggested above. File names are case-sensitive. Please generate the required zip file by directly compressing all the above files, rather than compressing a folder containing those files.

Example:

Suppose Bob with student ID 1234567 and Alice with student ID 7654321 are in a team in Session H1. They should submit a zip file named “1234567-H1.zip”, in which it contains “readme”, “members”, ‘Makefile” and all the necessary source files to build the final executable programs. The “members” file should be a file with two lines, the first line being “1234567” and the second line being “7654321”.

A Appendix: Marking guideline

Code:	
One correct implementation:	1
Additional optimization on request processing:	1
Speed:	1
Report:	
Clear description of implementation:	1
Testing and verification:	1
Performance discussion:	4
Presentation:	1
Total:	**10**