Parallel/Distributed Computing (CSEG414/CSE5414) Assignment #2

2020. 11. 10

Due Date: Nov. 30th (Monday)

1. (150 Points) In this problem, you are required to implement a text-based chatting program using *Apache Kafka* (https://kafka.apache.org/intro), a publish-subscribe based event streaming platform. You can use Kafka installations over any platforms (e.g., Windows, Linux, Mac, etc.). The tutorial for the installation will be provided with this document. Please use the <u>default configuration values</u> for Kafka and Zookeeper (*Apache Zookeeper* is needed to run Apache Kafka: https://zookeeper.apache.org/) to develop this application. Here are the detailed instructions to build this application.

< Requirements for building a chatting program >

1. The chatting program consists of 3 sub-windows (**Login Window**, **Chatting Window**, and **Chat Room Window**), where each window has its own menus explained below.
2. This program starts from the Login Window.
3. Even if the program restarts, it should preserve its previous states.

■ Login Window Menu (2 menus)

1. ***Log in*** – Receives a user ID (an alphanumeric string with maximum 32 characters) and moves to **Chatting Window** with the ID (no validation is required).
2. ***Exit*** – Ends this program.

■ Chatting Window Menu (4 menus)

1. ***List*** – Lists the chat room names created by users on the screen.
2. ***Make*** – Creates a new chat room.
3. ***Join*** – Joins a chat room with chat room name (an alphanumeric string with maximum 32 characters) and moves to corresponding **Chat Room Window**. (Joining a chat room which is not created beforehand is not allowed)
4. ***Log out*** – Goes back to the **Login Window**.

■ Chat Room Window (4 menus)

1. ***Read*** – Read newly created messages from the topic and display them on the screen. When a message is read, the message will be popped up and is not displayed again for next read request.
2. ***Write*** – Write text in a chat room. The written text is recorded using the topic of the chat where the key is the user ID used in the Login Window, and the value is the written text.
3. ***Reset*** – Moves the consumer offset to the beginning so that the messages can be read from the beginning.
4. ***Exit*** – Go back to the **Chatting Window**.

< Sample > Here is a sample run of a chatting program.

```
Welcome to CacaoTalk
1. Log in
2. Exit

cacaotalk> 1 <ENTER>
cacaotalk> ID: joey <ENTER>

Chatting Window
1. List
2. Make
3. Join
4. Log out

cacaotalk> 2 <ENTER>
cacaotalk> Chat room name: sogang <ENTER>
"sogang" is created!
cacaotalk> 1 <ENTER>
sogang
cacaotalk> 3 <ENTER>
cacaotalk> Chat room name: sogang <ENTER>

sogang
1. Read
2. Write
3. Reset
4. Exit

cacaotalk> 2 <ENTER>
cacaotalk> Text: Hey, Dooly! <ENTER>
cacaotalk> 1 <ENTER>
joey: Hey, Dooly!
cacaotalk> 1 <ENTER>
cacaotalk> 3 <ENTER>
cacaotalk> 1 <ENTER>
joey: Hey, Dooly!
cacaotalk> 4 <ENTER>

Chatting Window
1. List
2. Make
3. Join
4. Log out

cacaotalk> 4 <ENTER>
```

2. (150 Points) In this problem, you are to design and implement a simple web server for static pages. In order to evaluate the performance of the server you have developed, you are asked to develop a (multi-threaded) client program that generates random requests to the server. Finally, you will be required to report the performance of your web server based on various traffic conditions from clients and server architectures used to implement your server. Read the instructions and the requirements very carefully.

< Requirements for building a web server >

1. The server will only support simple HTTP requests and responses (GET <path>, followed by a carriage return and line feed) where the response is the body of the requested document followed by connection termination.
2. The server must implement minimal failure functionality (i.e., return a "404 Not Found" instead of just dropping the connection).
3. The server should be runnable with a parameter determining the port on which it listens for connections.
4. The server will follow the *master worker model with thread pool* discussed in the class and the number of worker threads created at startup is a command line parameter. In this architecture, one master thread (I/O thread) is responsible for receiving all network I/Os and delivers the requests to the workers via a single queue. Each worker thread monitors the queue (each worker thread needs to synchronize to access the queue) and process each request.
5. The default location of the HTML files should be on the local disk (i.e., /var/tmp/<your user name>).
6. There is an existing implementation that you can easily get and modify for the tedious parts of HTTP protocol. Try to refer to the *micro_httpd* (http://www.acme.com/software/micro_httpd/). The size of this code is only 200 lines of C code.

< Requirements for building a client >

1. The client program simulates the concurrent requests from multiple browsers and it should be multi-threaded.
2. The number of threads and how many times each thread accesses your server are command line parameters at startup.
3. The interval between two consecutive requests within a thread is either random or uniformly distributed.
4. Assume that the set of files the client can access is stored in a file and the name of the file is given as a command line parameter.
5. In the end, your program should report how many bytes it got from the server.

< Requirements for the performance evaluation and the report >

1. Test your web server using your client program.
2. Vary the number of threads in both client and server and report your observations as well as the performance of the server.
3. Vary also other parameters (e.g., size of file, interval, etc) and report your

observations as well as the performance of the server.

4. Be careful of the caching effect. Since you are making repeated requests, your files are likely to be cached. Check the performance of your server when the files are cached, thereby a little disk I/O may occur. How much is the performance benefit in this environment ? Try to make your server having substantial I/O and check the performance benefits. Report your observations.
5. Try to modify the client and server codes so that they maintain the connections (*persistent* connections) as long as each client has something to send to the server. In this case, the server should be modified to follow *peer model with thread pool* and each worker uses *epoll* mechanism (https://en.wikipedia.org/wiki/Epoll) to monitor multiple connections. Compare the performance with original implementation by varying various parameters and report the results as well as how you have implemented client/server codes.

< How to submit >

Create a **tar** file with the name of **"pdc2_학번.tar"** where it contains **two** zip files for each problem (e.g., problem1.zip, problem2.zip). Each zip file contains all the source files, makefile, readme file, and related documents for each problem. For problem 1, generate a zip file using **"File> Export> Project to Zip File"**. Finally, upload the **tar** file to Sogang Cyber Campus.