**CSCD467/567 Homework 4 and 5 (project)**

**Distributed Web Services, Thread Pool and Management on Server, Parallel Client Query**

Turn in all your source files and a readme.txt file that describes the commands used to compile and run your program in a **terminal** window. Please zip all your files into a .zip file and name it as firstNameInitial + Lastname + hw45.zip, submit the zip file on canvas.

Note that this homework is considered a mid-term project, which is worth 200 points and considered as two pieces of homework when calculate the total weighted final grades.

**What is provided**

In the start-up source folder, you are provided two java source files, CapitalizeClient.java and CapitalizeServer.java. The provided code implements a client and a server on the TCP level using sockets. Based upon the demos and the discussion in class about the provided program, you have to modify and add more features to the server and the client program. Basically, the **provided** client sends a string to the server, and then the **provided** server converts the received letters into all capitalized letters. Afterwards, the server sends the capitalized string back to the client for display.

**Problem description**

The client and server programs already support concurrent queries and are able to handle multiple simultaneous connections or requests. However, when each client connection (request) is detected, the server has to create a new thread to handle that connection, which is not very efficient. The reason is that creating a new thread for each request has overhead. **Instead,** we like the server to have a pool of running threads (threadpool) that have already been created when the server starts up. The threadpool is ready to process the incoming connections whenever they are received over the network. If there is no connection is coming in, all threads in the threadpool will wait and be blocked until connections arrive. After a thread in the threadpool processes a connection request, if there are more requests to process, the thread will grab the next available request in the job queue to process.

In addition to the pool of Worker threads and the main server thread, there is another thread called **ThreadManager** on the server, which is created and always runs while the server’s main thread runs. The purpose of the ThreadManager is to monitor both the workload of the job queue for the threadpool and the number of available threads in the threadpool for use. In particular, when the number of jobs in the job queue waiting for the threadpool to process exceeds a threshold **T1** but less than or equal to **T2** (T2 > T1)**,** the ThreadManager will double the number of threads in the threadpool, and maintain that number of threads as long as the number of jobs in the job queues stays in the range [T1, T2], inclusive of T2. If the number of jobs in the job queue to be processed exceeds the threshold **T2**, the number of threads in the pool is doubled again and maintains that number as long as the number of jobs is greater than T2 but less than the capacity of the job queue. In this case, we assume we have enough thread holders in the threadpool for the new threads that we will add dynamically. In this project, when the server is first started, it has 5 threads in the threadpool by default. We will use the threshold values T1=10, T2 = 20 and the max capacity of the job queue is 50. And the maximum number of thread holders is 40.

If the number of jobs in the job queue drops as the connections decrease, the ThreadManager halves the number of threads in the pool by following the opposite criteria as we discussed above.

The following table summarizes the growth and shrinkage policies of the threadpool. If Nj, the number of jobs waiting to be processed in Q, is less than or equals to 10, you suppose to have 5 threads in the pool. If Nj becomes bigger than 10 as more connections arrive, you have to increase the number of threads(Nt) to 10, doubling the previous value of Nt.

|  |  |  |  |
| --- | --- | --- | --- |
| (Nj)-Num of Jobs in Q | Nj<=T1(10) | T1(10) < Nj <= T2(20) | T2(20)<Nj<Capacity(50) |
| (Nt)-Num of Threads | 5 | 10 | 20 |

**Requirements**

1. You have to implement a ThreadPool class on your own; you CANNOT use any Java built-in threadpool classes or objects. Your ThreadPool class will contain a pool of Worker threads. Your design might look like the following **pseudo code**:

**Class ThreadPool {**

**int maxCapacity; //maximum number of threads in the pool**

**int actualNumberThreads;**

**WorkerThread holders[]; //stores the worker thread references**

**boolean stopped; //used to receive a stop signal from main thread**

**MyMonitor jobQueue; //shared by all WorkerThread in the pool and ThreadManager**

**//and the main server thread**

**Inner class WorkerThread {**

**//each Worker will grab a job in the jobQueue for**

**//processing if there are available jobs in the jobQueue.**

**…**

**}**

**public … startPool() {**

**//start all available threads in the pool and Worker**

**//threads start to process jobs**

**…**

**}**

**public …increaseThreadsInPool(…) {**

**//double the threads in pool according to threshold**

**…**

**}**

**public …decreaseThreadsInPool(…) {**

**//halve the threads in pool according to threshold**

**…**

**}**

**public …stopPool(…) {**

**//terminate all threads in the pool gracefully**

**//all threads in pool terminate when a command KILL is sent through the client // to the server.**

**…**

**}**

**public …numberThreadsRunning(…) {**

**…**

**}**

**public …maxCapacity(…) {**

**…**

**}**

**…and other methods as you need.**

**}**

1. ThreadManager will be constantly polling the status of jobQueue and the status of ThreadPool every **V** seconds in order to decide whether to increase or decrease or maintain the current number of the threads in the pool, according to the criteria discussed in the previous section. The ThreadManager terminates when a command “KILL” is sent through the client to the server.
2. Modify the client and server program so that it can accept and support five types of commands, including “ADD,4,5”, “SUB,10,9”, “MUL,2,3”, “DIV,4,2” and “KILL”. Noted that there is no space between operands and operators in the command string. When a message such as “ADD,4,5” is sent to the server, the server main thread has to create a job and put it into the jobQueue used by the ThreadPool object. Then the job will be processed by one of the running threads in the pool, which returns the result 4 + 5 = 9 to the **corresponding** client that made the original request “ADD,4,5”. That client will show the results for the original request in the GUI provided. **You have the freedom to design your job object stored in the jobQueue, but general information for a job should include which client the server is talking to and what service is requested etc.** The KILL command will terminate all threads in the server gracefully, including the main thread. The SUB, MUL and DIV commands mean subtraction, multiplication and division, respectively.
3. Print all (keep a log of) actions performed on the server onto standard out, E.g. “ThreadManager doubled number of threads in the pool at 11:34pm Oct. 23 2014, now total running threads in pool is 20.” Or “Worker thread id=1 processed service request ADD,3,4 at the time ……”.
4. When you first start your ThreadPool, you should have **5** threads running by default and ready to process jobs. The maximum capacity of the thread holder of the pool is **40**, and the maximum capacity of jobQueue is **50**. All other variables, **V, T1, T2** should be programmed as parameters so that you can easily change them for testing.
5. Write a parallel program **without** using the GUI components, in which you can simulate a large number of clients connecting to the server in a short period of time. In this test, you can slow down the speed of the WorkerThread in the pool by using sleep() method. So that you can clearly see the ManagerThread triggers the increaseThread() method of the ThreadPool object**. Show a picture or screen shot of your test for this scenario**, save the screen shot into the project report described below. For simplicity, in this parallel program, each client thread will send a **single** command to the server. Next, it displays the calculation result that is sent back from the server, then the client thread terminates.
6. Following step 6, after all your requests are processed – because there are less jobs in the queue eventually – you can see ManagerThread trigger the decreaseThread() method of the ThreadPool object. **Show a picture or screen shot of your test for this scenario.** Please save the screen shot into the project report described below.
7. Please design a test that shows when you send KILL commands to the server, the server gracefully shuts down. Please save the screen shot into the project report described below. KILL will gracefully shut all the server threads. **If a thread on the server is processing a job when KILL is received, you will have to terminate the thread after the thread finishes the current job processing.**
8. Write a project report to describe how you designed your system, (e.g. java classes, methods in each java class, how each class interacts with another? What issues did you run into and how did you solve them?). The project report should also include the test results described in items 6, 7, and 8. The project report should also include some normal cases where you send commands such as “ADD,4,5”, “SUB,10,9”, “MUL,2,3”, “DIV,4,2” to the server, as well some abnormal cases, like “ADD,3,” or “NOCOMMMAN, 4, 6”. The server Worker thread has to handle these abnormal cases gracefully without crashing.
9. Keep in mind your server main thread should always be running (no waiting state), otherwise clients that try to connect will encounter a hang-up. Particularly, when the jobQueue for the threadpool is full, i.e. there is no spot for a new job to be created and be enqueued, your server main thread in this case has to ignore the client’s request by sending back a message “The server is currently busy, please connect later!” and it then closes the socket for the new connection. Please design a test to verify this feature on your server, show a screen shot of your server log in your project report. You could use the same piece of code you used for item 6 for this test, or by manually creating jobs through the GUI window.
10. You can safely assume that a client connects to the server whenever there is a need. In other words, after a client has been connected to the server, the client sends its command to the server immediately right after that connection is established.
11. You are not allowed to use any features provided in the java concurrent package. I.e. only the topics we learned are allowed.
12. You can have your own design for any details that have not been specified in this document. In other words, if your programs fulfill all requirements listed in this documents, you are fine.
13. **Graduate students are supposed to fulfill all requirements specified above, while undergraduate students can skip step 6 and 7. If an undergraduate student successfully implements step 6 and 7, they will get 10 bonus points on top of 200 points.**