Scalability

Engineering Robust Server Software
Homework 4

For this assignment you will be studying performance and scalability of server software threading and work distribution strategies. For this homework, you may write in any language (or combination of languages) you wish.

The server will receive requests from clients and perform a small task before responding to the request. At a high-level, the server will create N buckets (e.g. via an array or vector). A client will send a request to add to the current value of a particular bucket. The server should use multi-threading to service requests, and you will evaluate the performance of both "create a thread per request" and "pre-create a set of thread" strategies. Each group will also have a 4-core VM to use for studying the scalability of your server performance across core counts.

A significant part of the assignment will consist of performance and scalability studies, as well as creating the client-side infrastructure to drive your performance tests. For example, you will need to stress your server code by sending client requests rapidly enough to saturate the server throughput.

The following outlines the operation of the server and its interaction with clients:

1. Your server will listen for incoming connections on port 12345. It will take one command line argument (an integer), which indicates a number of buckets to create.
2. A client makes a connection to the server and sends 1 request in the form of one line containing two comma-separated base-10 unsigned integer followed by a newline. The first number represents a delay count. The second number represents the bucket to add. For example: "5,10\n"
3. The server handles a request as follows:
    a. First, it assigns the request to a thread.
    b. Then the thread executes a delay loop. The "delay count" (first number) represents the number of seconds to delay. The following shows a C version of the delay loop. If you are using another language, you will need to port this code. Note the delay operation should not simply call sleep(…), as this is intended to mimic active CPU compute that a server might do to process requests.

```
struct timeval start, check, end;
double elapsed_seconds;

gettimeofday(&start, NULL);
do {
    gettimeofday(&check, NULL);
    elapsed_seconds =
        (check.tv_sec + (check.tv_usec/1000000.0)) -
        (start.tv_sec + (start.tv_usec/1000000.0));
} while (elapsed_seconds < req_delay);
```

c. After the delay loop, the server will add the "delay count" value to the "bucket" (second number). Don't forget to prevent race conditions in your multi-threaded code!
d. The last step to processing a request is to send a single line response back to the client consisting of a single base-10 unsigned integer followed by a newline. The number should contain the new value of the bucket that was updated for the request.

You should experiment, collect performance measurements, and analyze the following dimensions (make sure your clients are driving sufficient request load to the server in call cases):

- The throughput of your server code when running with 1, 2, and 4 cores available (this is the performance scalability of the code).
- Two threading strategies: (1) create per request and (2) pre-create.
- Small vs. large variations in delay count (with 4 cores active). For small delay count variability, you might try delays of 1-3 seconds. For large delay count variability, you might try delays of 1-20 seconds.
- Different bucket sizes (32, 128, 512, 2048) buckets (with 4 cores active).

Your deliverables for this assignment are as follows:

- Your server code, setup such that the TAs can run it in docker-compose

- A writeup, including graphs in which you analyze the scalability of your server. Please use good experimental methodology in creating these graphs (run the experiment multiple times, draw error bars, etc). The writeup must be in PDF format and named `report.pdf` inside a sub-directory called `writeup`.

- Your testing infrastructure, which you used to test your server (both for functionality and scalability). Your TAs should be able to reproduce your results (within the variance of normal experimental noise) from the test infrastructure you provide. This testing infrastructure should be in a sub-directory called `testing`.