CS6650 Fall 2018 Assignment 2

Overview - Wearable Device Data Management

This test scenario is modeled on the usage of wearable devices such as step counters. A user wears a device and it counts their steps progressively throughout the day. The test scenario models a single day of data uploads from a fixed user population, interspersed with requests to read the current step count from users.

The basic scenario models 24 hours of upload and read requests from a fixed user population, with parameters that can be varied to control the test duration.

Server Endpoints

The server endpoints supported are as described below.

POST /userID/day/timeInterval/stepCount
Where:
- userID ranges between 1 and user population size
- day ranges between 1 and number of days to upload for a test (default 1)
- timeInterval ranges between 0 and 23, representing the hour in a day
- stepCount is an integer between 0 and 5000

GET /current/userID
- Returns stepCount

Where:
- userID ranges between 1 and user population size
- stepCount is the cumulative number of steps for the most recent day stored for a user

GET/single/userID/day
- Returns stepCount

Where:
- userID ranges between 1 and user population size
- day is a specific day number that is stored for a user
- stepCount is the cumulative number of steps for the specified day

GET/range/userID/startDay/numDays
- Returns: stepCounts[]

Where:
- userID ranges between 1 and user population size
- day is a specific day number that is stored for a user

numDays is the number of days to returns step count totals for, including start day
stepCounts[] is the cumulative number of steps for each day specified and the total for all steps, ie {17900, 11234, 4900, 34024}

Step 1 - Implement the Server

Implement the required application interface in a Java server running on your free EC2 instance. The instance will need to store the data in a database. You will therefore need to create and launch an RDS MySQL or Postgres instance and enable your Java server to connect to this instance.

Test the server to ensure it stores and retrieves data correctly in your database.

Step 2 - Implement the Client

Implement a multithreaded client to test the performance of your server. Your client should be initialized, either through the command line or a properties file, with the following parameters:

MaxThreads: number of threads in peak phase, default 64
Server IP address: URL of server
Day number: day number to generate data for (default to 1)
User population: Default 100,000
Number of tests/phase: Default 100

Your test client has the same 4 phases as assignment 1, and in addition generates POST/GET requests for the following *timeIntervals* (simulating the progression of a day) within each phase:

1. **Warmup phase:** create 10% of the maximum number of threads. This phase generates requests from timeInterval 0-2.
2. **Loading phase:** create 50% of the maximum number of threads. This phase generates requests from timeInterval 3-7.
3. **Peak phase:** create the maximum number of threads. This phase generates requests from timeInterval 8-18.
4. **Cooldown phase:** create 25% of the maximum number of threads. This phase generates requests from timeInterval 19-23.

In each phase, first create the specified number of threads. Each thread should then execute (number of tests/phase * phase length) times. For example, in the warmup phase with default settings this would be (100*3), In each iteration execute the following sequence of actions/requests:

Generate random 3x(userID, timeInterval, stepCount) from valid ranges for the phase
POST /userID1/day/timeInterval1/stepCount1
POST /userID2/day/timeInterval2l/stepCount2

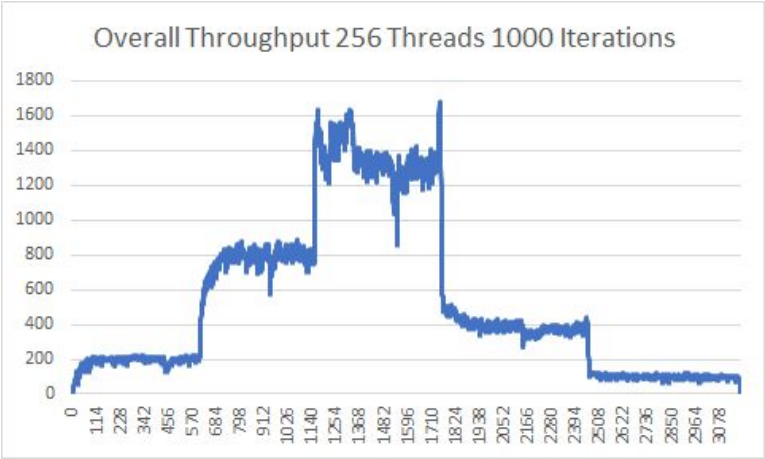
GET /current/userID1
GET/single/userID2/day
POST /userID3/day/timeInterval3/stepCount3

As per assignment 1, measure and record the timestamp and latency for every round trip to the server. You will want to design a mechanism to do this that is as lightweight as possible in terms of execution, so that the metrics capture does not slow down your request generation. Remember, you may potentially be capturing many millions of results for a single test run. Holding these in memory for the test duration is unlikely to be a good solution!

Test your client with the default settings for number of threads and tests/phase. Then postprocess the results to 'bucket' requests into one second intervals, and create a graph that plots each second interval for the test on the x-axis, and the number of requests processed that second on the y-axis. Also calculate the overall throughput for your test. You can do this in code or in say a spreadsheet. Whatever works.

If you implement your test phases as non-overlapping, this should be very obvious from your plot of latencies (why?). Ideally, test phases should overlap, so think about how this might be built into your client. Non-overlapping is fine, you just won't win prizes for your overall throughput (but no points lost!).

Regardless, your plot should look something vaguely like this in terms of overall shape:



Step 3 - Performance Testing

For this step you simply need to produce a plot like the one from the previous step, but for 4 different *maxThreads values*, namely {32, 64. 128, 256}. For each test, also show the average throughput and the p95 and p99s. To create an equal playing field for each test, delete the

contents of your database before launching your client. This ensures every test starts with an empty database.

The default server settings for number of threads in tomcat and number of database connections *should* be ok for running this test. It might be fun and profitable to try tuning these to see if you can get better performance from your server. We'll cover this issue in class.

Step 4 - Add Load Balancing

Configure EC2 Elastic Load Balancing to spread your client load across multiple server instances. Your test results from Step 3 should inform how you configure the load balancer in terms of number of instances and autoscaling. The tests with a low number of client threads are unlikely to trigger your scaling rules, but you may see this occur with 128/256 threads in your client.
Produce the same plots and statistics as Step 3, and write a sentence or 3 comparing the results from the two steps to try and explain the differences.

Step 5 - Bonus Points

Simple - scale things up against your load-balanced server configuration. Your ultimate aim is 1000 threads and 1000 iterations. This should trigger load balancing rules!

Plot the results for the largest test your achieve.

Submission:

As in assignment 1, submit a pdf to blackboard with a URL to your repo and the following

Steps 1 and 2 (20 points)
The plot and performance statistics showing the results for a test run with the default client settings for threads/tests.

Step 3 (5 points)
Plots and statistics for the 4 tests runs against a single server.

Step 4 (5 points)
Plots and statistics for the 4 tests runs against your load balanced servers. Briefly state your load balancer setup and compare the results against those you produce for step 3.

Step 5 (2 bonus points)

The plot for the largest test you were able to complete.

Deadline: Tuesday 30th October Noon