gortonator / bsds-6650 Public

Code Issues 1 Pull requests 1 Actions Projects Security

master

bsds-6650 / assignments-2021 / Assignment-1.md

gortonator Update Assignment-1.md ✓ History

3 contributors

173 lines (127 sloc) | 11.4 KB

CS6650 Fall 2021 Assignment 1

Overview

You work for Upic - a global acquirer of ski resorts that is homogenezing skiing around the world. Upic ski resorts all use RFID lift ticket readers so that every time a skiier gets on a ski lift, the time of the ride and the skier ID are recorded.

In this course, through a series of assignments, we'll build a scalable distributed cloud-based system that can record all lift rides from all Upic resorts. This data can then be used as a basis for data analysis, answering such questions as:

- which lifts are most heavily used?
- which skiers ride the most lifts?
- How many lifts do skiers ride on average per day at resort X?

In Assignment 1, we'll build a client that generates and sends lift ride data to a server in the cloud. The server will simply accept and validate requests, and send an HTTP 200/201 response. In Assignment 2, we'll add the processing and storage logic to the server, and send a richer set of requests. In Assignments 3 and 4, we'll get a little crazy, so make sure you lay solid code and design foundations in the first two assignments!

Implement the Server API

The initial server API is specified using Swagger

In this assignment you need to implement this API using a Java servlets. Each API should:

1. Accept the parameters for each operations as per the specification
2. Do basic parameter validation, and return a 4XX response code and error message if invalid values/formats supplied
3. If the request is valid, return a 200/201 response code and some dummy data as a response body

Good simple illustrations of how to handle JSON request payloads are here and here.

This should be a pretty simple task. Test each servlet API with POSTMAN or an equivalent HTTP testing tools.

Make sure you can load the resulting .war file onto your EC2 free tier instance you have created and configured in lab 1 and call the APIs successfully.

Build the Client (Part 1)

This is the major part of this assignment. We want a multithreaded Java client we can configure to upload a day of lift rides to the server and exert various loads on the server.

Your client should accept a set of parameters from the command line (or a parameter file) at startup. These are:

1. maximim number of threads to run (numThreads - max 256)
2. number of skier to generate lift rides for (numSkiers - max 100000), This is effectively the skier's ID (skierID)
3. number of ski lifts (numLifts - range 5-60, default 40)
4. mean numbers of ski lifts each skier rides each day (numRuns - default 10, max 20)
5. IP/port address of the server

In addition, each ski day is of length 420 minutes (7 hours - 9am-4pm) from when the lifts open until they all close.

Based on these values, your client will start up and execute 3 phases, with each phase sending a large number of lift ride events to the server API.

Phase 1, the *startup* phase, will launch numThreads/4 threads, and each thread will be passed:

- a start and end range for skierIDs, so that each thread has an identical number of skierIDs, caluculated as numSkiers/(numThreads/4). Pass each thread a disjoint range of skierIDs so that the whole range of IDs is covered by the threads, ie, thread 0 has skierIDs from 1 to (numSkiers/(numThreads/4)), thread 1 has skierIDs from (1x(numSkiers/(numThreads/4)+1) to (numSkiers/(numThreads/4))x2
- a start and end time, for this phase this is the first 90 minutes of the ski day (1-90)

For example if numThreads=64 and numSkiers=1024, we will launch 16 threads, with thread 0 passed skierID range 1 to 64, thread 1 range 65 to 128, and so on.

Once each thread has started it should send (numRunsx0.2)x(numSkiers/(numThreads/4)) POST requests to the server. Each POST should *randomly* select:

1. a skierID from the range of ids passed to the thread
2. a lift number (liftID)
3. a time value from the range of minutes passed to each thread (between start and end time)

With our example, if numRuns=20, each thread will send 4x(1024/16) POST requests.

The server will return an HTTP 201 response code for a successful POST operation. As soon as the 201 is received, the client should immediately send the next request until it has exhausted the number of requests to send.

If the client receives a 5XX response code (Web server error), or a 4XX response code (from your servlet), it should retry the request up to 5 times before counting it as a failed request.

Once 10% (rounded up) of the threads in Phase 1 have completed, Phase 2, the *peak phase* should begin. Phase 2 behaves like Phase 1, except:

- it creates *numThreads* threads
- the start and end time interval is 91 to 360
- each thread is passed a disjoint skierID range of size (numSkiers/numThreads)

As above, each thread will randomly select a skierID, liftID and time from the ranges provided and sends a POST request. It will do this (numRunsx0.6)x(numSkiers/numThreads) times. Back to our example above, this means phase 2 would create 64 threads, and and each sends 12*(1024/64) POSTs.

Finally, once 10% of the threads in Phase 2 complete, Phase 3 should begin. Phase 3, the *cooldown phase*, is identical to Phase 1, starting 25% of numThreads, with each thread sending (0.1xnumRuns) POST requests, and with a time interval range of 361 to 420.

When all threads from all phases are complete, the programs should print out:

1. number of successful requests sent
2. number of unsuccessful requests (ideally should be 0)
3. the total run time (wall time) for all phases to complete. Calculate this by taking a timestamp before commencing Phase 1 and another after all Phase 3 threads are complete.
4. the total throughput in requests per second (total number of requests/wall time)

You should run the client on your laptop. Thus means each request will incur latency depending on where your server resides. You should test how long a single request takes to estimate this latency. Run a simple test and send eg 10000 requests from a single threadto do this. You can then calculate the expected throughput your client will see using Little's Law. If your throughput is not close to this estimate for each of the test runs, you probably have a bug.

Building the Client (Part 2)

With your load generating client working wonderfully, we want to now instrument the client so we have deeper insights into the performance of the system. To this end, for each POST request:

- before sending the POST, take a timestamp
- when the HTTP response is received, take another timestamp
- calculate the latency (end - start) in milliseconds
- Write out a record containing {start time, request type (ie POST), latency, response code}. CSV is a good file format.

Once all phases have completed, we need to calculate:

- mean response time (millisecs)
- median response time (millisecs)
- throughput = total number of requests/wall time
- p99 (99th percentile) response time. Here’s a nice article about why percentiles are important and why calculating them is not always easy.
- max response time

You may want to do all the processing of latencies in your client after the test completes, or you may want to write a separate program to run after the test has completed that generates the results. Check your results against those calculated in a spreadsheet.

The client should calculate these and display them in the output window in addition to the output from the previous step, and then cleanly terminate.

Submission Requirements

Submit your work to Blackboard Assignment 1 as a pdf document. The document should contain:

1. the URL for your git repo. *Make sure that the code for the client part 1 and part 2 are in seperate folders in your repo*
2. a 1-2 page description of your client design. Include major classes, packages, relationships, whatever you need to convey concisely how your client works.Include Little's Law throughput predictions.
3. Client (Part 1) - run your client with 32, 64, 128 and 256 threads, with numSkiers=20000, and numLifts=40. Include the outputs of each run in your submission (showing the wall time) and plot a simple chart showing the wall time by the number of threads. This should be a screen shot of your output window.
4. Client (Part 2) - run the client as per Part 1, showing the output window for each run. Also generate a plot of throughput and mean response time against number of threads. Again, this should be a screen shot of your output window.

Bonus Points

Charting

It is usually interesting to plot average latencies over the whole duration of a test run. To do this you will have to capture timestamps of when the request occurs, and then generate a plot that shows latencies against time (there's a good example in the percentile article earlier). You might want to plot every request, or thinking ahead for assignment 2, put them in buckets of, for example, a second interval, and plot the average response time for that time interval bucket.

Grading:

1. Server implementation working (5 points)

2. Client design description (5 points) - clarity of description, good design practies used
3. Client Part 1 - (10 points) - 1 point per run output, 1 point for the chart, 5 points for sensible results! Points deducted if actual throughput not close to Little'sLaw predictions.
4. Client Part 2 - (20 points) - 1 point per run, 1 point for the chart. 5 points for calculations of mean/median/p99/max/throughput (as long as they are sensible). 10 points for wall time within 5% of wall time for Client Step 1.
5. Bonus Points: Up to 3 bonus points

Additional Useful Information

Building Swagger Client with Java 11

You need to modify your POM, add the following dependencies:

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.2.11</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
    <version>2.2.11</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>2.2.11</version>
</dependency>
<dependency>
    <groupId>javax.activation</groupId>
    <artifactId>activation</artifactId>
    <version>1.1.1</version>
</dependency>

<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

Problems with GSON import in .war filr in Intellij

Sometimes there's an issue building the GSON jar file into a servlet, such that when the servlet is deployed it fails because GSON is missing.

Try adding the gson jar to your project from the Maven website

For IntelliJ you also need to:

1. Go to project structure
2. choose the artifacts tab
3. select the gson maven package and put it into the lib directory of the artifact.

Compile and deploy ... and cross your fingers and toes!!

For more details check this out

Randon number generation in Java Threads

Worth a read for random number generation in your client ;)