CS6650 Fall 2017

Assignment 2 - Building the Server

Overview

In this course we will progressively build a distributed system that handles significant request loads and data volumes. In this second assignment we focus on building the server for a new application. We'll test this server out by extending client framework you built in assignment 1.

After your stellar performance in assignment 1, you are headhunted by the rapidly expanding ski resort. Blackler-Whistcomb. The ski resort is installing RFID ticket readers at all lifts. When a skier (snowboarders are banned) wants to get on a ski lift they place their RFID-enable lift pass next to an RFID reader. It checks the pass is valid and then records:

- Resort ID: String (numeric)
- Day Number: (1-365 representing the operational day in the season of the resort, eg opening day is Day 1)
- Timestamp: String (0-360 representing number of minutes after lifts open (9am) until they close (3pm)
- Skier ID: String (unique, numeric)
- Lift ID: String (1-40)

The resort expects an average of 40K skiers per day, and on average each skier will ride 20 lifts between 9am and 3pm when the lifts close. At 3pm, all the RFID readings are transmitted to a central file system in the resort, merged and then loaded into the skier database which is hosted by a cloud provider.

From the database, skiers can use a mobile app or web browser to see how many lifts they rode each day, how many vertical metres they skied, and various other statistics that keep skiers very happy.

Skiers always finish skiing at 3pm and go to a bar. It takes roughly 10 minutes to buy a beer and then everybody will look at their ski app to see how many vertical meters they skied that day. It is therefore vitally important to have the data available within 10 minutes of lifts closing so everyone can brag about their ski day!

You task in this assignment is to:

1. Build the server interface and database that can be used to load data and support skier queries.

2. Modify your client from assignment 1 to submit the data from the ski lifts to load into the database at the end of every ski day, and make skier stats available.

Step 1 - Build the Server

Create a server that accepts two HTTP requests:

POST \load\{ resortID, dayNum, timestamp, skierID, liftID)
GET \myvert\skierID, dayNum)
    Returns {Total vertical, number of lift rides)

Implement simple test methods that send a simple responses (no server business logic required yet) and deploy your server locally to test from your browser or IDE-supplied testing tool.

Step 2 - Build the Client

Next we want to build a client to test our server under load. You will be supplied with a file containing one day of skier data. Your client should read every record in this file and send them to the server POST method.

Doing this in a single thread is probably the best place to start to test the functionality works. This however is not likely to be the fastest solution. Hence you might want to think about strategies to use multiple threads to send requests to you server concurrently. Be creative :). Make sure whatever strategy you use is parameterized so you can choose how many threads to start from the command line.You should instrument your client to get the same measures as in Assignment 1 for every test, as these will guide your experimentation later.

You should also add charting capabilities (Assignment 1 extra points) to your client. 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). Generating data in a format that can be loaded into a speadsheet or a tool like JMeter is probably the best bet. You can use Java charting libraries like JFreeChart if you want to get some experience with such things. They are fun.

Test the client against the simple server from step 1. Do this with your server running on AWS to introduce latencies. Don't worry about optimizing performance at this stage, as the server is soon going to behave very differently! Just make sure your client-side multithreading strategy can obtain high-ish throughput from your server and your measurements and charts give you good insights into how the system is behaving.

Step 3 - Add Persistence to the Server

Now you have validated that you can send all the data in a ski day to your server successfully, it's time to add a persistent tier. Or in other words, a database! The AWS RDS service is likely to

be your friend here, at least initially. If you want to experiment with other options (e.g AWS Aurora, DynamoDB, etc), you are more than welcome. Just beware of the costs involved.

The data model is pretty simple. You need to store:

- All daily ski activity records (ie what the client sends you each day)
- Data about each skier (basically skierID) and the ski lifts and total vertical they ride every day they ski. An average skier will ski 40 days every winter and 20 lifts per day.

In this assignment you are only required to support the query to get a summary of a skier's data (as per GET in Step 1) for a given ski day (at the moment you just have 1 day so it's easy. At the moment ….).

Remember, a key quality criteria is a very rapid response to this request. Your boss knows a happy skier is one with their first beer in one hand and their daily ski stats in another. This makes people come back and spend more money on beer, which is the major profit maker for any ski resort.

You can assume:

- There are only 40,000 skers, so you can create a database entry for each and assume it exists prior to loading data
- There are only 40 ski lifts. Lifts 1-10 rise 200m vertical. Lifts 11-20 are 300m vertical. Lifts 21-30 are 400m vertical, and lifts 31-40 and 500m vertical. As you can probably guess, this data doesn't change very often :)

One you have built and tested the database tier and server, try loading the whole test data set from the client (don't worry about performance yet) into the database. Then for every skier issue a GET request as a test to make sure everything is logically correct. This will probably take a while.

You should also make sure you can easily delete all the database contents generated on a single day so you can return the database to a known state. This will be very useful for performance tuning later in this assignment. You should be able to create a script to do this.

Step 4 Throughput Experiment - Optimizing Write Performance

Once again, Step 4 is when the fun starts :).

As luck would have it, your boss is Scottish. This has two major effects on your work life. First, you can't understand anything she says, so you have to communicate with her via email/Slack to understand what she wants. Second, she is extremely conservative with her budget, and wants you to do everything possible to meet performance needs at minimal cost. You tell her that scaling out the server would almost certainly improve performance, but with only 1 ski resort

to manage, she wants you to do everything possible to operate with a single (free) server resource.

Your task in this step therefore is to experiment with your client and server threading to see what the best performance level is that you can attain for loading a day of ski data.

You basically have three main parameters to 'tweak':

1) The server thread pool size (configured in apache/glassfish config files)
2) The database connection pool size (depends on your database config)
3) The number of client threads that send POSTs simultaneously (configured in your client code)

Typically the relationship between these values is:

#Client threads > server thread pool size > database connection pool size.

Remember this use case is writing to the database. Writing is slower than reading as you can't exploit caches effectively.

You task is to experiment with these settings to try and achieve the best throughput you can in terms of loading the data. Throughput is simply #requests/walltime, but in this experiment the number of requests is fixed, so your aim is to load as fast as possible. Return the database to its initial (mostly empty) state between tests to ensure the starting conditions are always the same.

Submit tables/charts/screen shots from test runs showing the throughput of the configurations you tested and the resulting performance, and any supporting explanations. Max 4 pages including charts.

Step 5 - Optimizing Read Performance

The use case here is based on 40K skiers trying to download their daily ski statistics from the bar at approximately 3.10pm each day.

Due to bar service delays, people don't all try to get their ski stats at 3.10. Blackler-Whistcomb is a big ski resort and there are 100 bars, each serving 400 skiers (!!!). We can model this traffic load by having 100 threads executing concurrently in the client, each sending 400 GET requests. Partition the skierID key space across these 100 threads so that every skierID is used to call GET once (total 40K requests).

Report the same statistics as in Assignment 1. You may want to fiddle with your server thread pool and database connection size for this experiment. You are only reading data so this should be a lot quicker than writing (and hence be more amenable to additional concurrency).

Submit screenshots of you best test run and charts showing latencies as per assignment 1.

Step 6 - Reading and Writing Together

Finally it is time to put this all together.

The use case we will model is what occurs at 3pm on Day 2 of the season. Basically:

- A new ski data set is uploaded (this will be provided) for Day 2
- All skiers decide to remind themselves what their stats were yesterday in anticipation of seeing their new numbers in 10 minutes after 3.10pm

From your perspective, this is just running the load tests from step 4 and 5 simultaneously. Just make sure your database contains all the data loaded for Day 1.

How you do this is up to you. For example, you may have all the code in one client that runs both the read and write loads in two thread collections. Or you may want to separate the write and read loads into two clients that can be run simultaneously. Refactoring into two clients is recommended, but not 100% needed right now. Hint: Two clients can be run in 2 command shells, potentially on different machines. Maybe you can start to see where this is heading ...

Submit screenshots of the one/two clients showing your performance statistics/charts and successful completion.

.

Grading and Submissions

There are 30 points available for this assignment, plus 2 bonus points.

Submit a pdf file to blackboard for assignment 2 containing:

1) A 1 page overview of your design (a simple block diagram would suffice). The aim is to quickly summarize your design so emphasize important components and abstractions. **(3 points)**
2) URL to your git repo. We'll be giving you detailed instructions on repositories and code quality metrics - watch this space **(5 points. We'll assess code quality)**
3) Step 4: Submit tables/charts/screen shots from test runs showing the throughput of the configurations you tested and the resulting performance, and any supporting explanations. Max 4 pages including charts. **(12 points)**
4) Step 5: Submit screenshots of you best test run and charts showing latencies as per assignment 1. **(5 points)**
5) Step 6: Submit screenshots of the one/two clients showing your performance statistics/charts and successful completion. **(5 points)**
6) The 5 submissions with the highest throughput for Step 6 will receive **2 bonus points.**

Deadline Monday 23rd October 11.59am PST