CS6650 Fall 2018

Assignment 1 - Building the Client

Overview

In this course we will progressively build a distributed system that handles significant request loads and data volumes.

In this first assignment we focus on building a client that can be used to generate load on a server and measures performance. This will give you excellent experience with multithreaded code and the basics of cloud server infrastructures.

Steps 1 to 3 are basically about getting the tooling and infrastructure set up correctly. They require very little code. Don't leave these until the last minute though as installation and configuration has many hidden traps for you to fall down. Get these steps right ASAP, and rest should be (more) straightforward.

Step 1 - Build the Server

Create a server that accepts HTTP request and sends trivial responses. In Java you will want to use JAX-RS or maybe the Apache HTTP libraries.

You'll probably want to use your favorite IDE to build the server with two methods supporting HTTP GET and POST. Below is an extract from the Netbeans-generated code for this server. Your IDE may look different but treat the below as a specification for your server's HTTP interface.

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String getStatus() {
    return ("alive");
}

@POST
@Consumes(MediaType.TEXT_PLAIN)
public int postText(String content) {
    return (content.length());
}
```

Deploy your server locally and test from your browser (POSTMAN) or IDE-supplied testing tool.

Step 2 - Build a Simple Client

Next we want to build a Java client to test our server. Again, feel free to use your IDE to help generate a JAX-RS client. Here's an extract from the Netbeans-generated client class, which you simply instantiate and call. It'll probably look different in your IDE. Run your client and test it connects to your server on *localhost* and successfully sends requests.

```
public <T> T postText(Object requestEntity, Class<T> responseType) throws
     ClientErrorException {
        return
             webTarget.request(javax.ws.rs.core.MediaType.TEXT_PLAIN)
                    .post(javax.ws.rs.client.Entity.entity(requestEntity,
                    javax.ws.rs.core.MediaType.TEXT_PLAIN),
                    responseType);
    }

    public String getStatus() throws ClientErrorException {
        WebTarget resource = webTarget;
        return
             resource.request(javax.ws.rs.core.MediaType.TEXT_PLAIN).get(String.class)
       ;
    }
```

Step 3 - Run Server on an AWS Instance

First, create and deploy ***an AWS free tier instance***. Amazon Linux is a simple OS choice but choose whatever you are familiar with. Make sure you install the ***jdk8*** if it's not there (hint - use the yum package manager on Amazon Linux).

You then need to decide how to deploy your server. Again you have choices depending on your development path:

- Install Glassfish if you used Netbeans
- Install tomcat if you used maven
- Probably others …

If you used a maven project for Step 1, this should be straightforward. If not, it should be trivial to modify your code to use Jersey as the Web Server. Here's 'hello world' with Jersey and maven which you can use as a guide.

Build your server in your IDE, and deploy the resulting .war file to your AWS instance, Here's how you do it on Tomcat.

Modify your client to point to the AWS instance IP address and web server port (Tomcat is by default 8080) and check it works just like it did on your laptop.

When it does, party all night, get some sleep and move on! Now for the fun part.

Step 4 Load Generating Client

This is where things get trickier :).

Your aim is to built a client that can accept four arguments, either on the command line or through a properties file (supply location on command line if so):

1. Maximum number of threads (default to 50)
2. Number of iterations per thread (default to 100)
3. IP address of server
4. Port used on server (default to 80 or 8080 depending on your Web server choice)

Upon starting, the client should process the command line and based on the **maximum number of threads**, execute the following 4 load testing phases. In each phase all threads perform identical work, namely they iteratively call the two HTTP endpoints in your server for the specified number of iterations.

1. **Warmup phase:** create 10% of the maximum number of threads and execute the specified number of PUT/GET iterations.
2. **Loading phase:** create 50% of the maximum number of threads and execute the specified number of PUT/GET iterations.
3. **Peak phase:** create the maximum number of threads and execute the specified number of PUT/GET iterations. Create the maximum number of threads and execute the specified number of PUT/GET iterations.
4. **Cooldown phase:** create 25% of the maximum number of threads and execute the specified number of PUT/GET iterations.

As an example, if the client is run with 20 threads and 100 iterations,.in the warmup phase, there will be 2 threads. In the loading phase, there will be 10 threads. In the peak phase, there will be 20 threads. In the cooldown phase, there will be 5 threads. Every thread will call the GET/POST endpoints 100 times and then twerminate.

Upon completion, the client should print out the total run time (wall time) for all threads in all phases to complete. ***The client should only terminate when all threads have completed in all 4 phases.***

Your output should look something like the below for each run:

```
>client 50 100 serverIPaddress 8080
Client starting …..Time: nnnnn
Warmup phase: All threads running ….
Warmup phase complete: Time nn.n seconds
```

```
Loading phase: All threads running
Loading phase complete: Time nn.n seconds
Peak phase: All threads running ….
Peak phase complete: Time nn.n seconds
Cooldown phase: All threads running ….
Cooldown phase complete: Time nn.n seconds
=======================================
Total number of requests sent: nnn
Total number of Successful responses: nnnn
Test Wall Time: nn.n seconds
```

Run with 20 threads and 100 iterations to check the synchronization and communications works fine. Next test out your solution with 100 client threads and 100 iterations

Submit screenshots of the two runs (20/100, 100/100) to validate that your code works.

Step 5 Adding Measurements

We want our client to inform us about the latencies it is experiencing for each request. To do this, you need to:

1. Measure the latency of **every** request sent to the server and successfully processed.
2. Collect the latencies for every call from every thread somewhere until all the threads are complete.
3. After completion of all client threads, process the latencies to generate the desired statistics.

The statistics you should produce are as follows:

- Total run time (wall time) for all threads to complete (as previous step)
- Total Number of requests sent
- Total Number of successful requests
- Overall throughput across all phases (total number of requests/total wall time)
- Mean and median latencies for all requests
- 99th and 95th percentile latency

Here's a nice article about why percentiles are important and why calculating them is not always easy.

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. You can also do the processing in a spreadsheet tool, but beware this might not be a suitable solution for later assignments (it is fine for this one though).

To validate your client, first run with 20 threads and 100 iterations.

If this works, run with 100 threads and 100 iterations.

Submit screenshots of the results from the two (20/100, 100/100) runs. Simply add these stats above to the test output in the style of the previous section of the assignment.

Step 6 - Build Server using AWS Lambda

Lambda is a serverless web processing layer. This means you don't have to build and manage your own servers. It has other advantages too, such as autoscaling.

Your aim here is to build a lambda server that has the same interface as you EC2 server instance. Here's a tutorial on building and deploying a Java lambda function using Eclipse. It should be possible with other IDEs too. but might take some Internet digging (post useful links on piazza for participation points!!).

Once you have the server running, point your Java client at it and run the same tests as step 5, and produce the same output, ie:

Submit screenshots of the results from the two (20/100, 100/100) runs. Submit exactly the same results as Step 5.

Step 7 - Bonus Points

You will get extra credit for either (or both) of the following:

Break Things :)

Experiment with the number of threads your client can support and report when something breaks. Perhaps with 1000 clients your server will overload an IP buffer somewhere, or requests will take so long they will timeout and fail? Or depending on how you are capturing latencies, your client may start throwing OutOfMemory exceptions?

Stress testing is fun and instructive. It's a skill you should be taking away from this course, so here's a chance to practise!

Charting

It is usually interesting to plot average latencies over the whole time 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 for that time interval bucket.

This is certainly a piece of functionality you will find useful as the course progresses (that might be a hint :))

Grading and Submissions

There are 30 points available for this assignment, plus 2 extra points if you want to undertake optional tasks.

Submit a pdf file to blackboard for assignment 1 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. **(1 point)**
2) URL to your git repo. **(3 points. We'll assess code quality)**
3) Two screenshots for step 4 showing correct execution and completion of the two specified tests **(4 points for each)**
4) Two screenshots for step 5 showing correct execution and completion of the two specified tests against your EC2 server instance. If you use an additional tool like a spreadsheet, show the results in this in addition to the two screenshots showing the test running **(12 points)**
5) Two screenshots for step 6 showing correct execution and completion of the two specified tests against your AWS Lambda server. If you use an additional tool like a spreadsheet, show the results in this in addition to the two screenshots showing the test running **(6 points)**
6) Step 7: Stress testing: Submit a short (1 page?) report describing what you did to explore the tolerances of your application, what broke it, and how. **(1 point)**
7) Step 7: charting: Submit a 1 page report detailing your test run, method of calculation, and chart showing latencies. **(1 point)**

Deadline Sunday 30th September 11.59pm