Project 5: Web Benchmark

Overview

An important part of any Web application development is to provision our Web site for the expected load from the users. To do this, we need to be able to measure how many concurrent user requests a single Web server can handle, so that we can estimate how many server instances are needed to handle the expected load. It is also important to be able to run any complex data processing tasks on a cluster of machines available to us, so that we can provide a near-time feedback from our user data to improve our application.

In this project, we run experiments to measure the number of concurrent requests that a system can handle.

Development Environment

The main development of Project 5 will be done using a new docker container named “locust” created from “junghoo/locust”. Use the following command to create and start this new container:

```
$ docker run -it -p4040:4040 -p8089:8089 -v {host_shared_dir}:/home/cs144/shared -
```

Make sure to replace `{host_shared_dir}` with the name of the shared directory on your host. The above command creates a docker container named `locust` with appropriate port forwarding and directory sharing.

Our new container has Locust pre-installed. You can check its version by the following command.

```
$ locust --version
```

As before, the default username inside the container is “cs144” with password “password”.

When you perform the tasks for Project 5, you also need to run the servers you developed in Projects 2 and 3. Make sure that your servers in the “tomcat” and “mean” containers are still available and run well by executing the following sequence of commands inside the *host*:

```
$ docker start tomcat
$ docker start mean
$ docker exec -d mean npm start --prefix {your_blog_server_dir}
```

Replace `{your_blog_server_dir}` with the path to your Project 3 directory inside the mean container, e.g., `/home/cs144/shared/project3/blog-server`. Make sure that your Tomcat server is available at http://localhost:8888 and your node server is available at http://localhost:3000 on your host machine.

By default, each container runs in its own isolated environment and can reach other containers only if it knows their IP addresses. In this project, we want to make the containers reachable by their *container names*, not just by IPs, so that we can send HTTP requests between containers more easily. This can be done by creating a custom *bridge network* and connecting our three containers to the bridge network. Run the following command inside the *host* to create a custom bridge network, named `cs144-net`:

```
$ docker network create cs144-net
```

Once a bridge network is created, any running container can be connected to it through the `docker network connect` command:

```
$ docker network connect cs144-net locust
$ docker network connect cs144-net tomcat
$ docker network connect cs144-net mean
```

The above sequence of commands will connect our three containers, “locust”, “tomcat”, and “mean”, to the bridge network “cs144-net”. (Make sure that the three containers are already running before you execute the above commands.)

Now the three containers can reach each other through their names. To verify, execute a shell in the “locust” container and run `curl` commands to issue HTTP requests to other containers like the following:

```
$ docker exec --user cs144 -it locust /bin/bash
cs144@e6a142ac3bf9:~$ curl http://mean:3000
cs144@e6a142ac3bf9:~$ curl http://tomcat:8888
```

The first `docker exec` command runs a shell in the locust container, so that the next commands are executed from the container, not from the host. The next two `curl` commands, therefore, issue HTTP requests from the locust container to the mean and tomcat containers, respectively. As long as the mean and tomcat containers have started and are running, you should see correct responses from the servers of the other two containers. Make sure that this is the case by checking the responses. From now on, you should use the `http://mean:3000` and `http://tomcat:8888` within locust container to access other containers.

Once you finish checking the containers and their network connectivity, you can stop them using the following command:

```
$ docker stop locust
$ docker stop mean
$ docker stop tomcat
```

From now on, all three containers can talk to each other through their names!

Testing Performance of Server(s)

Now your task is to (1) learn how to use Locust to test the performance of your server, (2) write some customized files that represent user behavior, (3) record the results of test cases, and (4) find the maximum number of users that the server could handle under some performance requirements.

Learn to Use Locust

You **must** go over the following Locust tutorial, step-by-step, before moving on:

- Quick Locust Tutorial

Before moving forward, make sure you **KILL the simple server** so that the request you send later is not responded by the fake server.

Mock Data Preparation

Before we start load testing our servers using Locust, we need to load some mock data into our databases. We prepared two scripts, mock_data_tomcat.sh and mock_data_node.sh for this purpose. Download the tomcat script **inside** the "tomcat" container and execute it like the

following:

```
$ wget http://oak.cs.ucla.edu/classes/cs144/project5/mock_data_tomcat.sh
$ bash ./mock_data_tomcat.sh
```

Do the same for the node script **inside** the "mean" container:

```
$ wget http://oak.cs.ucla.edu/classes/cs144/project5/mock_data_node.sh
$ bash ./mock_data_node.sh
```

Now our databases have been populated with 500 fake blog posts by the user `cs144`.

Write Locust Files

Once we have all the test data inserted, it's time for you to write locust files to load test the servers. In particular, you are required to write 6 files to perform the following tests. In all tests, set the `wait_time` to be between 0.5 section to 1 second:

Locust File for Tomcat Server

1. *read_tomcat.py*
    - In this file, we are simulating the scenario where all requests from users are read intensive. The user whose name is `cs144` would randomly open one of his posts via `/editor/post?action=open&username=cs144&postid={num}`, where `{num}` is a random postid.
    - **Note**: In this test file, use `/editor/post?action=open` as the `name` parameter for `client.get()`, so that requests with different `postids` in the URL are merged into a single entry. Also, make sure that postid that user opens should be between 1 and 500. Since our user "cs144" only has 500 posts, he will get nothing otherwise!
2. *write_tomcat.py*
    - In this test file, we are simulating the scenario where the requests from users are write intensive. The user `cs144` would modify one of his posts randomly by changing the title to "Hello" and the body to "***World!***" at `/editor/post` with the body

`action=save&username=cs144&postid={num}&title=Hello&body=***World!***`. Replace `{num}` with a random number between 1 and 500.

- **Note**: Use `/editor/post?action=save` as the request `name` parameter. You may use *only* POST method here since GET method is not supported.

3. *mixed_tomcat.py*

   - In this test file, we are simulating a more realistic scenario where some users are reading posts while others are writing. In this test, 20% of users are write intensive while the remaining 80% are read intensive.
   - **Note**: Just "cut and paste" the tasks/functions you wrote in previous files and combine them with different weights. 20% of user tasks must come from the "write task" defined in *write_tomcat.py* and the remaining 80% of user tasks must come from the "read task" defined in *read_tomcat.py*.

Locust File for Node Server

4. *read_node.py*

   - In this test file, we are simulating a similar behavior of `read_tomcat.py` except that now we are testing the performance of our Node.JS server. The user `cs144` would randomly open one of her posts via `/blog/:username/:postid` API. Again, remember to limit the postid between 1 and 500 when you pick one randomly.
   - **Note**: Use `/blog/cs144` as the `name` parameter for this request.

5. *write_node.py*

   - In this file, we are going test the server performance under write intensive requests as we did in `write_tomcat.py`. The user `cs144` randomly update one of her posts by changing its title to "Hello" and body to "***World!***" through `/api/posts`. Remember that you need to obtain the JWT authentication token through the login page before calling this API. Limit the postid to between 1 and 500.
   - **Note**: Use `/api/posts` as the `name` parameter for the request.

6. *mixed_node.py*

- In this file, we combine the read intensive tasks and write intensive ones in a single file as you would expect. The percentage would remain the same as 20% write and 80% read.
- **Note**: Just reuse the tasks/functions you wrote in previous files and combine them with different weights.

Note: In the request URLs of your locust files, please do not include the hostname and just use the absolute path, like `/blog/username/3`. The hostname and port will be provided as a command line parameter when you run the files as described below.

Run Locust Files and Save Results

Part 1

Once you fish writing the six locust files, run Locust without the web UI using the following setting for `read_tomcat.py` and `write_tomcat.py`,

```
$ locust -f {your_locustfile} --host=http://tomcat:8888 --headless -u 10 -r 10 -t
```

and the following setting for `read_node.py` and `write_node.py`.

```
$ locust -f {your_locustfile} --host=http://mean:3000 --headless -u 10 -r 10 -t 30
```

Once they finish running, fill in the performance.json file with your results, where the field values are 0. More precisely, you should fill in the fields, "`aggregated # reqs`", "`aggregated req/s`" and "`response time for 98% of the requests`", with the numbers that you get from the two final summary tables of your tests.

Note: The filled values should be **numbers**, not a string. For example, if your aggregated req/s is 160 req/s, just put the number `160`, **NOT** `"160"` or `"160 req/s"`. Our grading script won't correctly recognize any value other than numbers. Do **NOT** change any other part of the performance.json file.

Part 2

After filling in the performance.json, you should be now familiar with the testing procedure. Let's try to find the maximum users that the server could handle under 20%-write and 80%-read load using `mixed_tomcat.py` and `mixed_node.py` files. In particular, we require that **the servers must return at least 98% responses within less than 1000ms for each request** (except login requests). That is, each URL name group except `/login` should return 98% of responses in less than 1000ms. Your task is to find the maximum number of users (in the unit of hundreds like 300 if the number if higher than 100, in the unit of tens like 60 if the number if higher than 10, and in the unit of 1 if it is below 10) that the servers can handle under this requirement.

Once you identify the maximum user number, rerun your test with the maximum number, but this time **with the `--only-summary` option**, so that only the summary output is printed. Save this output to `summary_tomcat.txt` (for the result from tomcat server) and `summary_node.txt` (for the result from the node server). You will need to submit these two files as your result for Part 2.

Note 1: Sometimes you might see fails caused by an error like `ConnectionError(ProtocolError('Connection aborted.', error(104, 'Connection reset by peer')),)`. It is very likely an indication that the number of users is more than that your server could handle, not necessarily something is wrong in your implementation.

Note 2: There is no right or wrong results for performance stats as long as your test files are correctly implemented. You just need to submit the results that you get from your performance benchmark.

Note 3: Now that you are done with Project 5, think about the performance of two servers in each scenario. Which one exhibited a better performance? Tomcat or Node.JS? Note that our comparison may be a bit unfair to the Node.JS server because (1) Tomcat did not "render" markdown to HTML when a blog post is "opened" and (2) Node.JS had to perform the extra authentication verification step for the "write" tasks. You may want to keep this difference in mind when you interpret your results. You do not need to provide an answer to this question in your submission. This question is to encourage you to think more deeply about what you observed from load testing.

What to Submit

For Project 5, you need to submit **a single zip file** named `project5.zip` that has the following packaging structure.

```
project5.zip
 +- read_tomcat.py, write_tomcat.py, mixed_tomcat.py
 +- read_node.py, write_node.py, mixed_node.py
 +- performance.json
 +- summary_tomcat.txt, summary_node.txt
 +- README.txt (optional)
```

All files should be contained directly under the project5.zip (without any enclosing folders). Each file or directory is as following:

1. read_tomcat.py, write_tomcat.py, mixed_tomcat.py, read_node.py, write_node.py, mixed_node.py: These are the six locust files to load test your tomcat and node servers
2. performance.json: This is the JSON file that contains the load test results.
3. summary_tomcat.txt, summary_node.txt: These are the summary table from the two mixed benchmark tests of Part 1
4. README.txt includes any comments you find worth noting, regarding your code structure, etc.

Please use our packaging script p5_package to package your submission. When you execute the packaging script like `./p5_package` in the directory that contains your Project 5 files, it will check for the required files, and package them into `project5.zip`.

Grading Criteria

- 6 test files runs without being terminated by an error
- `performance.json` contains your test results
- Outputs in the 2 summary files meet the requirements