ASSE - Exercise 8: Testing

Group 1

2. Performance Tests with JMeter

Performance testing on was carried out via Jmeter. We only tested the HTTP endpoints open to end users, namely the retrieving of a task from the task list and the creating of a task. The results of the performance tests can be found in the table below. We carried out a large number of performance tests with different parameters to start with to get a feeling for their effects. In the end we set up 9 different tests (thread groups), each having a different set of parameters (indicated by the Test name). Each test consisted of two types of requests: one GET requests that fetched a task from the task list and one POST request which created a new task in the task list. For each of these tests we noted down the error rate and the throughput. All services were restarted between each test in order to make sure they all started from the same point.

Test	Type of request	Error%	Throughput / sec
100 threads - 10s ramp-up - 300ms const delay	GET	0,00%	1,5
	POST	0,00%	1,5
100 threads - 1s ramp-up - 300ms const delay	GET	0,00%	8,8
	POST	0,00%	6,7
100 threads - 10s ramp-up - 30ms const delay	GET	0,00%	8,8
	POST	0,00%	8,0
250 threads - 10s ramp-up - 300ms const delay	GET	0,00%	18,2
	POST	0,00%	16,9
250 threads - 1s ramp-up - 300ms const delay	GET	6,40%	13,9
	POST	0,00%	14,4
250 threads - 10s ramp-up - 30ms const delay	GET	0,00%	20,8
	POST	0,00%	19,1
500 threads - 10s ramp-up - 300ms const delay	GET	0,00%	15,0
	POST	0,00%	13,7
500 threads - 1s ramp-up - 300ms const delay*	GET	0,00%	10,6
	POST	28,80%	0,4
500 threads - 10s ramp-up - 30ms const delay	GET	0,00%	10,9
	POST	0,00%	1,3

* Task list crashed

The test results in the table above indicate that the service does well until it starts having between 250 and 500 concurrent users adding and retrieving tasks. However, it must be noted that each thread (user) in our tests executed 2 requests in a relatively small time frame. It is not clear if this represents real user behaviour. Two tests resulted in the service struggling to respond and timing out. These both had a low ramp-up time so it is clear that the pressure from having a very short time between requests can overwhelm the server or the service. In the last two tests, when we got to 500 users with short times between requests, the trend was always similar. The first ≈400 GET requests and the first ≈100 POST requests would fly through and then everything grind to a halt. Then the server would start responding in batches, but the time between each batch did not seem to follow any trend. One test (indicated by *) managed to crash the task list. During this test, the responses followed the trend described above, until it stopped responding completely for 13 minutes (having responded to all 500 GET requests and 356 POST requests). Based on the internal server logs, the task list encountered the following connection error to the database a number of times over that period before restarting by itself:

```
Exception in monitor thread while connecting to server tapas-db:27017
```

In addition to these trends we made a number of anecdotal observations about the behaviour of the services and the server while running the tests:

- Having the logs running in a terminal SSH'd into the server seemed to have a significant effect on the results of the more intense tests.
- We accidentally ran some tests with an invalid task ID in our GET request. Interestingly, this seemed to have a positive effect on the results of the test. This was most likely because an error was being thrown early in the processing of the request, so we never got to the part where we retrieved the task from the database (as we are still using both an in-memory task list and the database, this will be changed).

Conclusions

It is not clear from our tests what the root cause the degraded response behaviour is. Based on some ad hoc testing, we found that when the service started taking a long time to respond, the server also became unresponsive to commands via ssh. Therefore it is possible that it is the server that our services run on that struggles to handle the large amount of requests coming in. Moreover, we only managed to crash our actual services once, and throughout our testing we did not manage to produce many internal errors. Further tests are required to find out if the server is a major bottleneck. However, we need to look into the connection errors that caused the task list to crash.

Submission note: We only noticed that saving the Summary Report files did not save the results after closing the program. The most important metrics from each report is however included in the table above (thank god).

Submission note II: You need to create a task and fill in the task-id and expected result for each GET request so that the assertions pass.

3. Chaos Engineering with Chaos Monkey

We used the chaos tool kit to do the tests. The experiment files can be found here:

Tasks list

- Kill/Restart:
  - Task list restarts normally.
  - During restart time, clients can't create new tasks. Also right now if during the down time a request is made to the task list from the auction-house or roster, the information is lost. —> This problem can be solved easy by adding at least two instances of the service. When adding another instance, we also need a load balancer in front of them so the incoming requests get handled by 1 instance. Because the tasks get saved to the database, each instance has access to the same data. Another approach to solve the problem about lost information from the auction-house and roster is to switch to an event driven communication with an event broker which can persist messages. In that case when the task-list is down and can not get the messages, the events don't get lost and as soon as the task-list starts again, the events get be handled.
- Latency:
  - Works still fine! The only disadvantage is that the loading times for the user are higher and therefore can make the user experience worse.

Roster:

- Kill/Restart:
  - Roster restarts normally.
  - Loads roster information from database on startup. If there is only one instance available for the roster we encounter the same problems about availability as in the task list. Solutions would also be the same.
- Latency:
  - Adding latency does not crash the workflows but having only one instance of the roster and high latency hurts the performance badly. This is because the roster is in the middle of the other services and therefore needs to handle more requests than the others.

Executor:

- Kill/Restart:
  - Executor restarts normally.
  - Currently there is no notification to the executor-pool when an executor crashes. This problem can be solved by adding health checks to the executor-pool so it can monitor about the health of executors.
  - When an executor get killed or restart unplanned and it was executing a task, the task is currently stuck and will not get executed again. After adding the previous mentioned health checks, the executor pool would notify the roster about the unexpected shutdown of an executor and the roster can remove the assignment of the task to the crashed executor and assign the task to a different executor.

- Latency:

- No problem with latency on the executors. They can take as long as they wan't for their tasks.

Global:

- Exception:
  - Getting random exception turns out to be a big problem in our system. When sending requests from one service to another and an exception occurs, the data is not getting resend or anything else. This means the information is just lost.
- Memory:
  - Memory tests seems to be fine. Running the application in the cloud will also help to mitigate this risk, as it can be configured that it get more memory if needed.