Documentation Week 6

Group 3

JMeter Tests

The performance tests were performed with the framework JMeter. We tested the creating and the retrieving of a task. Additionally, we tested the creating of an executor. The results of the performance tests can be found in the table below. We carried out performance tests with different parameters to start with. In the end we set up 3 different test each having 3 different set of parameters. For each of these tests we noted down the error rate, the latency, the deviation and the throughput. We configured a ramp up time of 1 second. All services were restarted between each test in order to make sure they all started from the same point.

Local Tests (1 sec ramp up time)

Tests	Error	Throughput	Latency	Deviation
POST – Add Task - 100	0%	23.0 / sec	194 - 3428	804
POST – Add Task - 1000	0%	32.4 / sec	1090 - 30065	8506
POST – Add Task – 5000	0% but stopped at 4071	42.7 / sec	2046 -91558	25263
GET – Get Task - 100	0%	90.0 / sec	35 - 258	55
GET – Get Task – 1000	0%	158.3 / sec	36 - 6119	1430
GET – Get Task – 5000	0% but stopped at 4070	217.4 / sec	44 - 18405	4061
POST – Add Executor - 100	0%	50.2 / sec	106 - 1170	279
POST – Add Executor - 1000	0%	88.3 / sec	421 - 10337	2881
POST – Add Executor - 5000	0% but stopped at 4098	42.7 / sec	1200 -35405	8207

Tests on VM before Roster Deployment (1 sec ramp up time)

Tests	Error	Throughput	Latency	Deviation
POST – Add Task - 100	0%	7.43 / sec	11633 - 13965	3007
POST – Add Task – 5000	0%	3.26 / sec	360’035 - 371’632	111’866
GET – Get Task - 100	0%	27.98 / sec	50 – 825	623
GET – Get Task – 1000	0%	34.67 / sec	2045 – 8125	3448
POST – Add Executor -100	0%	24.98 / sec	2187 – 3518	581
POST – Add Executor -1000	0%	32.35 / sec	24’964 – 32’546	4108

Tests on VM after Roster Deployment (1 sec ramp up time)

Tests	Error	Throughput	Latency	Deviation
POST – Add Task - 100	0%	5.7 / sec	3200 – 16’000	3547
POST – Add Task – 1000	0%	6.2 / sec	18’000 – 154’346	41’163
GET – Get Task - 100	0%	25.6 / sec	2201 - 3004	307
GET – Get Task – 1000	0%	33.3 / sec	1105 – 22’052	4160
POST – Add Executor -100	0%	33.4 / sec	740 - 2500	254
POST – Add Executor -1000	0%	33.1 / sec	15’000-29’000	3770

Main Concern

- In order to achieve accurate results of our experiments we decided to conduct our performance tests on our VM rather than testing the performance locally.

- Despite that, our Interactions with the VM were not reliable. On multiple occasions we had to stop testing with JMeter, because requests failed (on seemingly random times) or a batch of requests could not be processed.
- In the Add Task actions there are two services involved. The Tasklist service request to the roster service and waits for the response before the HTTP 200 response is sent back to the client. In the other two requests Add Executor and Get Task there is only one service involved and no further HTTP requests. Therefore we have high latency on the Add Task action.
- Due to non-parallel processes on the VM, the latency increases with higher number of requests per seconds. When there are only 10 requests per seconds the latency is around 100ms. We assume there is a bottleneck on the tomcat server or in the traefik reverse proxy, which redirect.
- In the table right below, we configured the ramp up time to find the threshold, where the queueing is starting to increase the latency with the "add execution" test. With a ramp up time of 40 seconds, there is still a latency of a few milliseconds. If the ramp up time is decreased to 30 seconds, the latency starts to increase unproportionally. Therefore, our Threshold is around 25 – 32 requests per second. A similar behaviour has shown in the "Get Task" Test.

Tests on VM after Roster Deployment (100 sec ramp up time -> 10 requests per sec)

Tests	Error	Throughput	Latency	Deviation
POST – Add Executor -1000 100 sec ramp up time	0%	10 / sec	38 - 274	13
POST – Add Executor -1000 40 sec ramp up time	0%	25 / sec	37 - 186	23
POST – Add Executor -1000 30 sec ramp up time	0%	32.7 / sec	42 – 3548	902

Key Takeaways

- Overall, we were surprised by the performance of our microservices.
- Expected load of our users could be handled, without any issues.
- Synchronous requests in our task list had a negative impact on our performance. Before the user could receive any feedback, the task list had to first contact the executor pool, which further blocked following requests.
- In order to make our services more scalable, we will implement asynchronous communication between both the executor pool and task list and additionally between the roster and executor pool.
- Unfortunately, our experiments could not be easily reproduced. JMeter detected latencies with a high deviation. This then prevented us from making precise statements.

Chaos Engineering

The Setup

In order to conduct chaos engineering, the dependency *Chaos Monkey for Spring* was used. In particular, the four different types of assaults *Latency*, *Exception*, *Memory*, *Killapp*, have been carried out. For the assaults, we have used JMeter, manual, and automated API calls to monitor the behavior of the tapas task microservice.

Latency (Request Assault)

```
"assaultProperties": {
"level": 1,
"deterministic": false,
"latencyRangeStart": 1000,
"latencyRangeEnd": 3000,
"latencyActive": true,
```

This assault adds random latency (1000 – 3000 milliseconds) to requests sent with JMeter. We observed that requests could still be handled, when we introduced latency.

Exception (Request Assault)

```
"exceptionsActive": false,
"exception": {
        "type": null,
        "arguments": null
},
```

By introducing the exception assault, the requests could be sent to the microservice but could not be handled by our tapas task list microservice. This could be counteracted by improving our exception handling within our tapas services.

Memory (Runtime Assault)

```
"memoryActive": true,
"memoryMillisecondsHoldFilledMemory": 90000,
"memoryMillisecondsWaitNextIncrease": 1000,
"memoryFillIncrementFraction": 0.15,
"memoryFillTargetFraction": 0.25,
```

Memory Assaults attack the memory of the Java Virtual Machine. When conducting this assault, we could not detect any observable changes to the behavior of the software. However, this can be explained as the documentation points out that this attack highly depends on the specific version of Java used.

Killapp (Runtime Assault)

For this type of attack, we have taken the advantage of the chaos toolkit to automate killing the app and subsequently disabling chaos monkey to go back to normal behavior. Please refer to the *killapp.json* in the submission files for further information.

Key Takeaways

- Despite our Virtual Machine not providing the greatest specs, we were still surprised by how our application could deal with adding latency to the requests and the rapid recovery from killing the entire services.

- We should implement better exception handling, as with the current state information about the requested task is lost.
- It is challenging to quantify the results obtained by chaos engineering, due to the lack of metrics. Similarly, without extensive configuration of chaos toolkit experiments, reproducibility of our chaos monkey tests was not possible to be achieved.

Responsibility of Group Members

Atilla	- Preparation of Presentation & Documentation - JMeter Testing on VM - Pair Programming of Executor-DB & Executor-Service & Executorpool-Service - Annotations of the Code
Fabian	- Admin Tasks, Integration, System & Chaos Testing
Fabio	- Implementation of executorpool Service and roster Service (mostly pair programming), Some adjustments in tasklist for roster implementation, Docker compose files
Phil	- Unit tests for executorpool - Untit tests with Mockito - JMeter tests local and VM - Report of performance testing
Valentin	- Implementation of the Executors (Joke and Compute) - Implementation of the Database connection in the Executorpool (Pair programming) - General coding support

ADRs and architectural characteristics

We discussed our architecture decisions as a team and documented them in ADRs all team members were involved in this process.