Software Engineering of Mobile Systems
Assignment 4

Table of Contents

Task 1 – Architectural prototype

Description: Develop an architectural prototype for the application of a performance tactic. You could find a performance tactic that consider the QAS (Quality Attribute Scenario) you already wrote as part of assignment 3...

Hand-in your results (e.g. graph or numbers) including type of AP (Architectural Prototype) developed and conclusions (e.g. does the tactic enable you to satisfy the QAS?, how large is the improvement when applying the tactic?

The scenario and aim of the architectural prototype is described briefly in the upcoming section. The overall scenario is based on an application with a web server that can handle user requests for filtering trajectory data. A web server takes the trajectory data as input and returns a filtered trajectory (e.g. mean or median).

Main questions that needs to be answered:

1. What is the latency with the use of a single node
2. What is the latency when scaling the amount of nodes (horizontal scaling)

Other interesting questions that can be answered:

3. What is the error rate of the requests single vs. multiple nodes?
4. How much network bandwidth is utilized single vs. multiple nodes?
5. How long does it take do execute a given scenario? Can it be reduced by introducing more nodes?

From Bass et. al: "*Performance is often linked to scalability—that is, increasing your system's capacity for work, while still performing well.* " In this case a load balancer is introduced to route the web traffic to multiple web servers. Thus, introducing a load balancer and multiple web servers into the system is the main concern with this architectural prototype.

The performance QAS is presented in the next section.

Performance tactic

“Performance is concerned with how long it takes the system to respond when an event occurs.”

Portion of Scenario	**Possible Values**
Source	A user wants trajectory data to be filtered (external)
Stimulus	Stochastic, within 1 minute
Artifact	Load balancer, Web server
Environment	Stress mode
Response	Filtered trajectory. Change in environment; Peak load (how many requests can the web server handle)
Response Measure	Throughput (number of requests) + Latency (time for response) + error rate

For this architectural prototype resources are managed by maintaining multiple copies of computation [Bass et. al., p. 139]. This choice is appropriate because the computation of trajectory data are mostly bound on processing time i.e. no data is stored to disk. It could be argued that concurrency could be introduced to increase response rate instead, but partitioning the trajectory data is assumed to be more time consuming for proposing a solution. Also, concurrency does not increase the amount of resources but instead aims at optimizing the utility of resources.

Test setup

The test setup consists of two physical devices and a local area network (LAN) with 1 GBit full duplex capacity. There are two different deployment settings; (1) single node scenario (illustrated on Figure 1: Deployment single-node), (2) multi-node scenario (illustrated on Figure 2: Deployment multi-node). The TestMachine is responsible for instrumenting the test run on the Server via the test suite artillery.io which can be used for load-testing and functional testing purposes. The Server hosts instances of the NodeJS trajectory filtering API. In the multi-node scenario the Server also hosts an Nginx based load balancer which balances the incoming requests in a least-connected fashion.

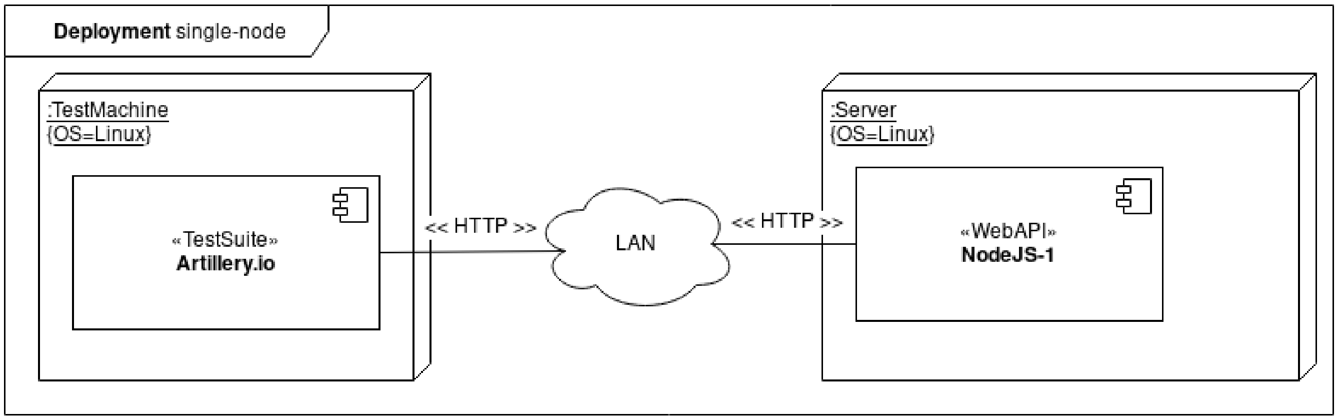

Figure 1: Deployment single-node

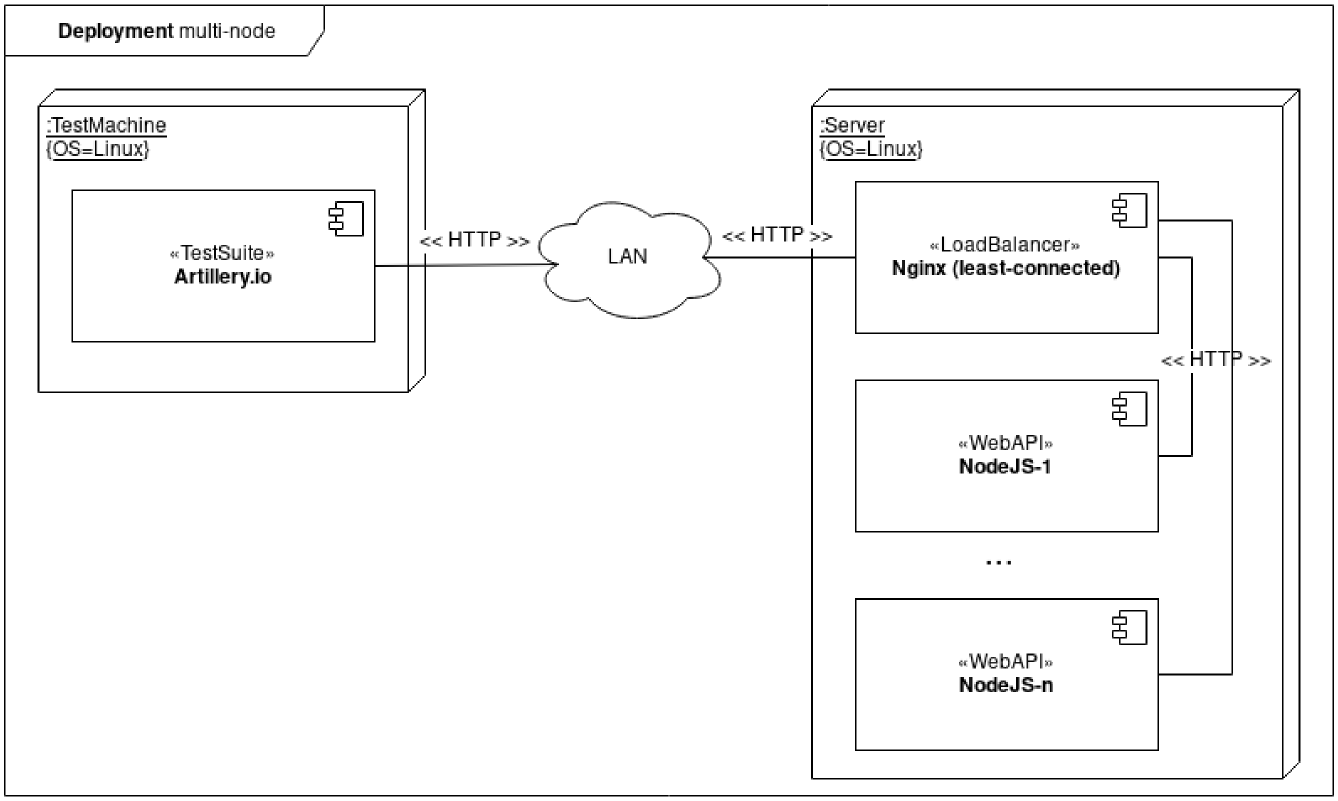

Figure 2: Deployment multi-node

From the client's point of view the server only exposes a single application on a specific port. The internal services are not exposed directly.

Table 1: Hardware specifications

TestMachine hardware specs	**Server hardware specs**
I7-4600U 2 cores 4 threads 2.10 Ghz (3.30 Ghz turbo) 12 GB RAM 1 Gbit LAN NIC	I7-6700K 4 cores 8 threads 4.00 Ghz (4.20 Ghz Turbo) 16 GB RAM 1 Gbit LAN NIC

Test method

Testing the architectural prototype was done with the use of artillery.io. The performance test was written in YAML to simulate 10 users arriving each second over a duration of 60 seconds. Each user makes 100 POST requests to the trajectory API. The payload of the requests contains *CSV data* along with the *interval* used by the mean and median trajectory filters. The payload is then processed by the API by converting it into GeoJSON with the filtered trajectory. The exact filtering method has a ~50% probability of being picked by artillery.io. The total amount of nodes are selected based on the testing server hardware specification. NodeJS runs on a single thread and the server have 8 threads available in which case this test only covers 8 nodes simultaneous.

The total number of requests created by the test is: 60*10*100 = 60.000 requests

Table 2: Artillery.io test

```yaml
config:
 target: 'http://192.168.0.13:3000' # Target server with the trajec
tory API
 phases:
  - duration: 60 # test duration
   arrivalRate: 10 # Number of new users every sec
 processor: "./request-builder.js"
scenarios:
 - name: "Mean API test"
  weight: 50 # Average 50% of users
  flow:
   - loop:
    - post:
      url: "/trajectory/mean"
      beforeRequest: setJSONBody
    count: 100
 - name: "Median API test"
  weight: 50 # Average 50% of users
  flow:
   - loop:
    - post:
      url: "/trajectory/median"
      beforeRequest: setJSONBody
    count: 100
```

Test results

Latency

This section answers the main questions first which is:

1. *What is the latency with the use of a single node*
2. *What is the latency when scaling the amount of nodes (horizontal scaling)*

Figure 3: Latency shows the latency results when scaling the amount of nodes available. The numbers on the figure indicates the 95 percentile. It is shown that when more nodes are added, the latency goes down but definitely not linear. Latency is therefore not the only interesting measure to examine.

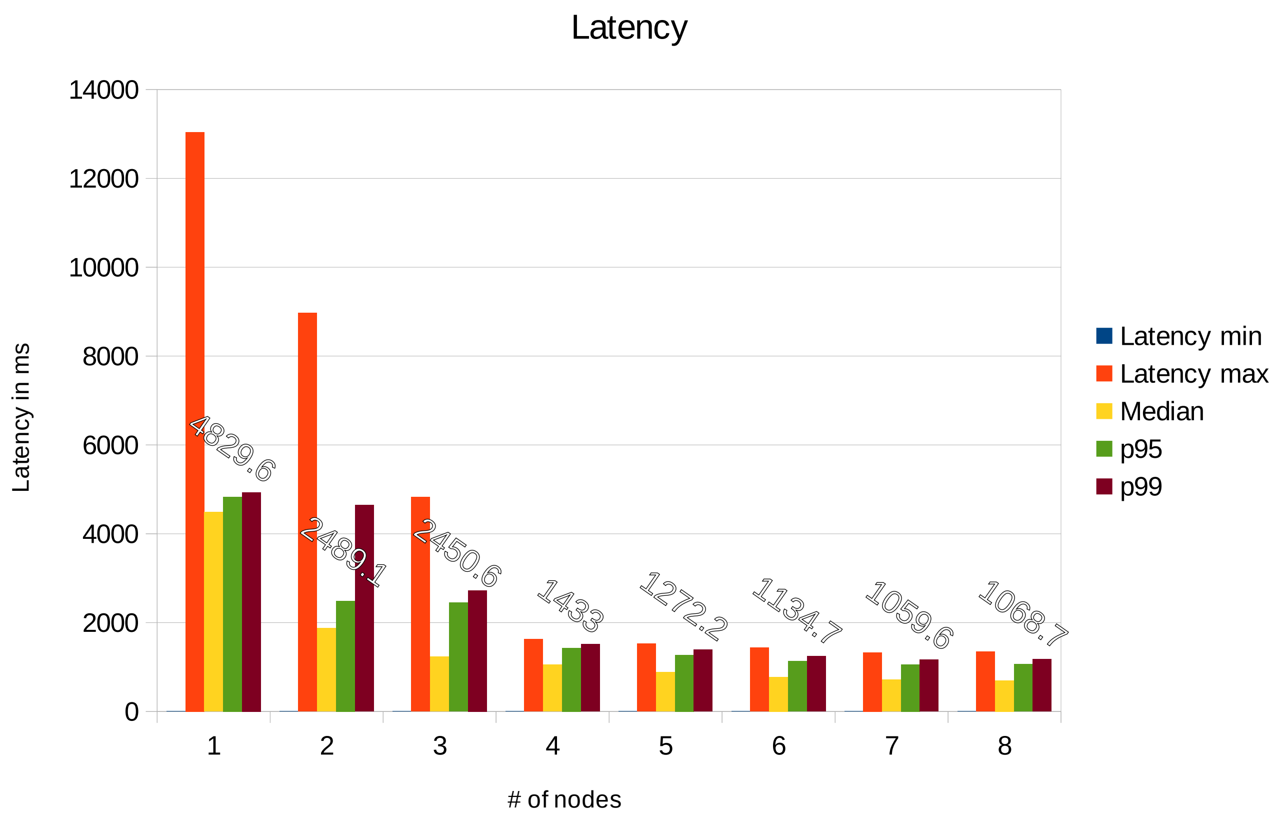

Figure 3: Latency

Error rate and success rate

This section answers the third question which was:

3. What is the error rate of the requests single vs. multiple nodes?

Figure 4: Success rate shows the success rate in percentages. All of the 60.000 requests goes through when only one node is used but it is at the cost of high latency. Up until the 5th node is added, the error rate is slightly unacceptable. This error rate could be explained by an occupied network and would require that several tests are run in order to know if this error rate is persistent. The figure shows that after the 6th node is added the success rate is acceptable.

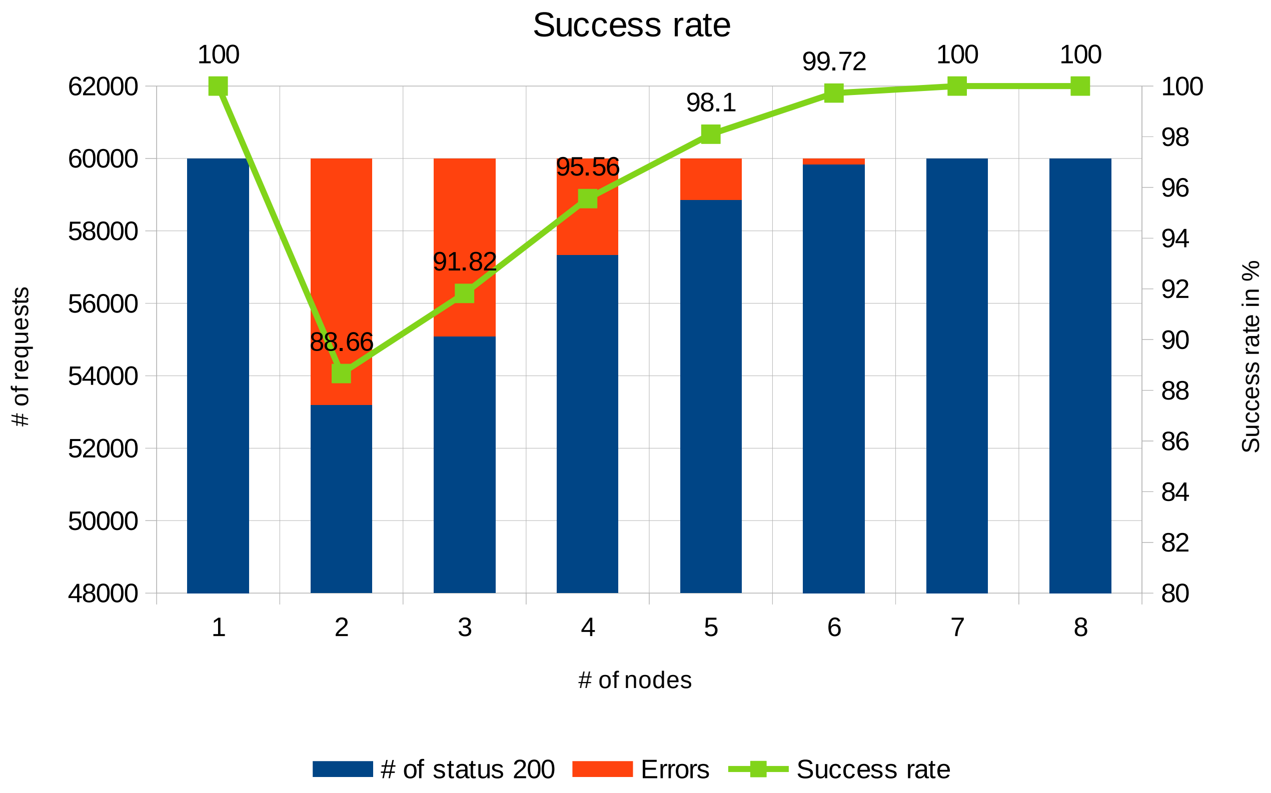

Figure 4: Success rate

Network bandwidth and requests per second

This section answers the fourth question which was:

4. How much network bandwidth is utilized single vs. multiple nodes?

Figure 5: Network utility shows average requests per second that the servers can handle, and also shows the average network bandwith utilization. It is visible that the requests scales very good as soon as just one node is added to the load balancer, but the scaling does not scale linear. After the 6th node is added, only a small benefit is obtained.

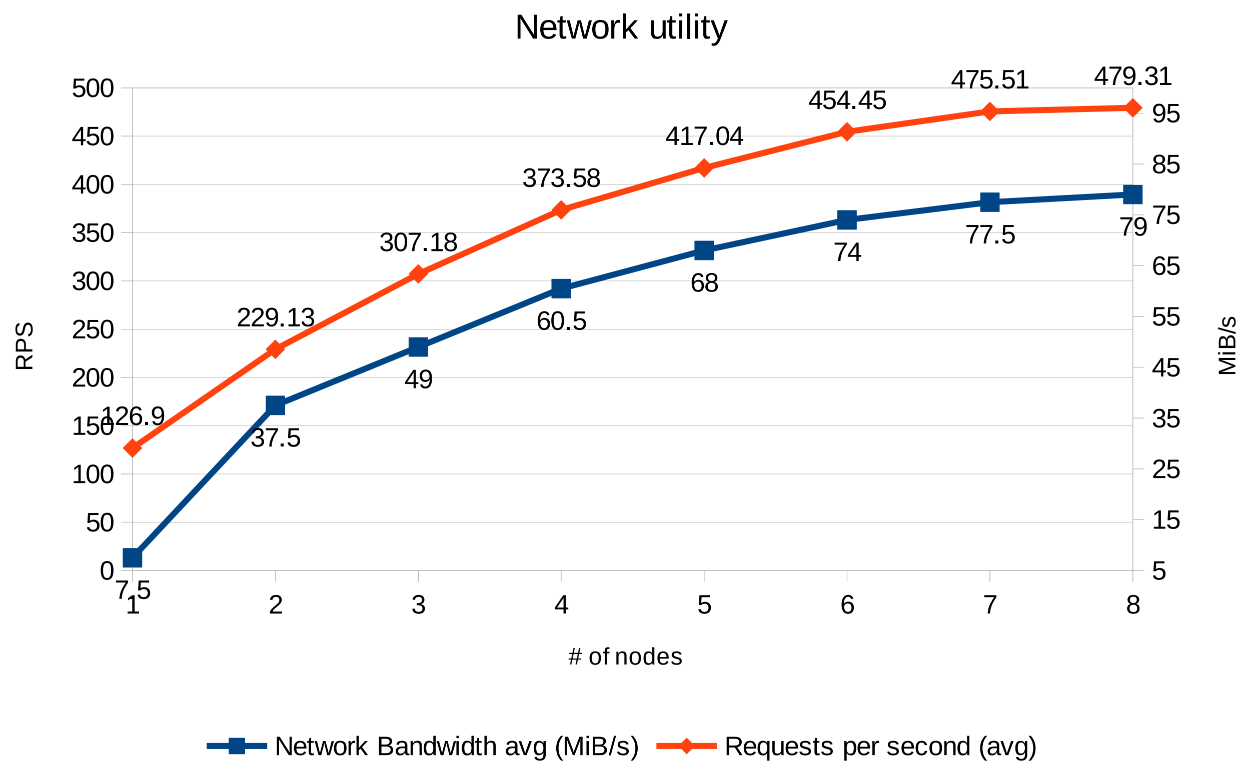

Figure 5: Network utility

Test duration

This section answers the 5th and last question:

> *5. How long does it take do execute a given scenario? Can it be reduced by introducing more nodes?*

Figure 6: Test duration shows that the duration to execute the 60.000 requests is drastically reduced as more nodes are introduced. When the 2nd node is added the duration of the test is actually reduced by 50%, when the 3rd node is added its 25%, the 4th node ~12%.

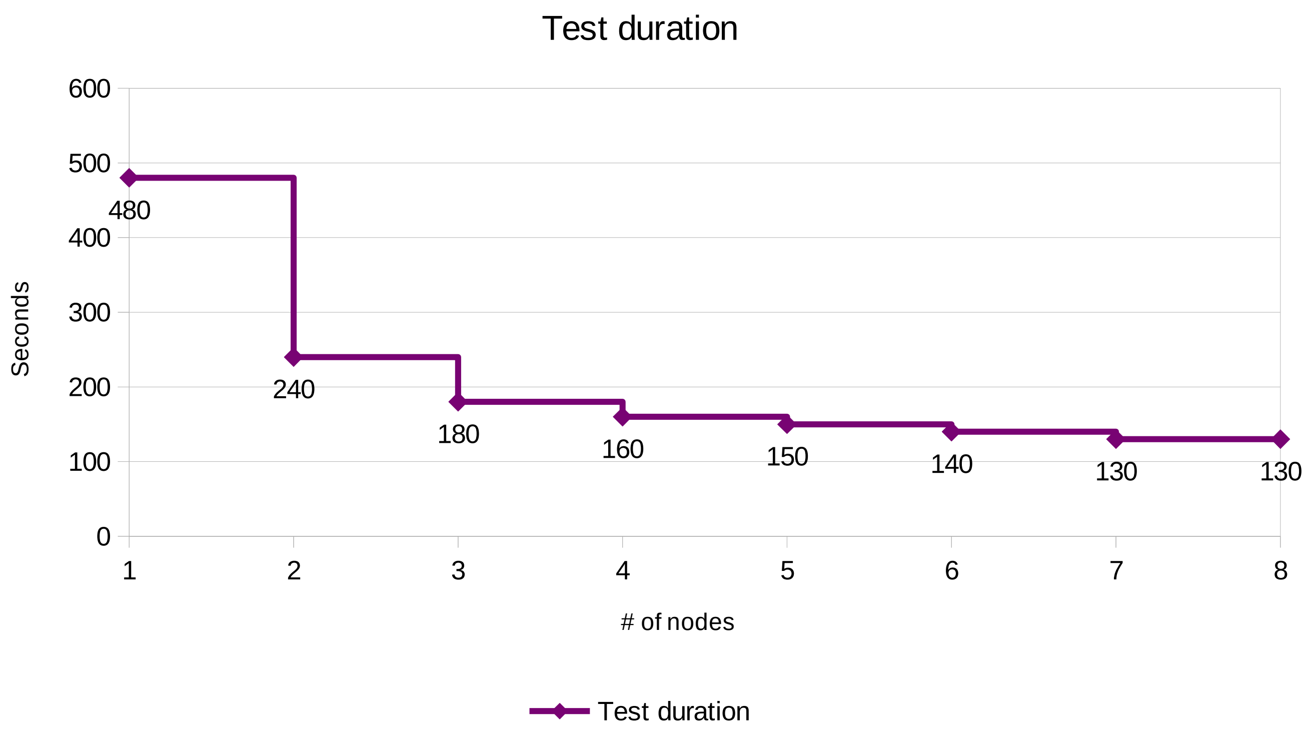

Figure 6: Test duration

Conclusion

By introducing a load balancer in nginx that uses least-connected method for balancing, and by scaling the amount of NodeJS servers available it has been shown that (1) latency was reduced by ~78%, (2) that the error/success rate could be kept at an acceptable level ~1% error rate, (3) that the RPS could be increased by ~374% at the cost of increased bandwidth, (4) that the time it took to execute 60.000 requests could be reduced by ~70%.

It is worth mentioning that the configuration should be tailored specific to the hardware at hand. In this case the sweet spot seems to be at 6-7 nodes + the load balancer. With this configuration it is possible to utilize all the 8 threads available on the server hardware.

Github repo

Contains all the source code for executing the prototype and scripts for automated artillery.io tests. The results from the tests are located in *tests/out/2019-11-03**:

https://github.com/csbc92/mobile-systems-assignment4