TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

Aleksei Gvozdev 176743 IAPM

PERFORMANCE TESTING OF AN ASYNCHRONOUS MULTI-AGENT SYSTEM

Master's thesis

Supervisor: Gunnar Piho
Docent

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Aleksei Gvozdev 176743 IAPM

ASÜNKROONSE MULTI-AGENT SÜSTEEMI KOORMUSTESTIMINE

Magistritöö

Juhendaja: Gunnar Piho
Dotsent

Tallinn 2019

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Aleksei Gvozdev

05.05.2019

Abstract

Nowadays web-based systems are used by thousands and millions of users. Large user base throughout the world provides superior possibilities for companies to deliver their products and services. However, alongside with possibilities large audience also brings its risks: the larger is the number of users, the higher is the load to the system, and the higher is the risk that the system will collapse under heavy load. In less severe but more common cases poor performance of the system results in long response time. As a consequence, this leads to dissatisfaction of end users, losing positions on the market, and diminishing incomes. Discovering such problems in production means higher cost of fixing. That is why before providing its services to the public, the business has to ensure that the system is ready for intensive usage. Performance testing is used for his purpose.

There is sufficient number of open source tools and case studies about performance testing of relatively simple monolith RESTful systems. However, these tools and case studies usually do not cover all aspects which become critical in performance testing of more complex systems. This is why sharing experience in this field is very important, and this is exactly what the current work is about.

This thesis is written in English and is 57 pages long, including 5 chapters, 22 figures and 4 tables.

Annotatsioon

Tänapäeval veebipõhised süsteemid pakuvad äridele häid võimalusi oma toodete ja teenuste müümiseks ning seetõttu on maailmas laialt levinud. Süsteemid erinevad kasutajate arvu poolest. Suurema kasutajate arvuga süsteemid toovad suuremat tulu. Kuid mida suurem on süsteemi kasutajate arv, seda suurem on tõenäosus, et süsteem ei pea koormusele vastu. See tähendab, et parimal juhul toimub, kas jõudluse langus või halvemal juhul süsteemi kokkuvarisemine. Tagajärjeks on lõppkasutajate rahulolu langus. Kuna töötava süsteemi vigade parandamine on lõppkasutajatele kulukas, siis, enne kasutusse andmist, tuleb veenduda, et süsteem suudab nõuetekohast piirkoormust taluda. Selleks kasutatakse koormustestimist.

Avalikes allikates on saadaval info suhteliselt lihtsate RESTful monoliitsüsteemide koormustestimiseks, kuid need lahendused ei ole sobivad keerukamate süsteemide jaoks. Kõnealuse töö kirjutamise eesmärgiks oli jagada oma kogemusi keerukamate ärisüsteemide koormustestimise kohta.

Lõputöö on kirjutatud inglise keeles keeles ning sisaldab teksti 57 leheküljel, 5 peatükki, 22 joonist, 4 tabelit.

List of Abbreviations and Terms

API	*Application programming interface* is a set of rules and constraints for interaction between two systems
Asynchronous	In programming, the process is asynchronous if it involves events happening independently of the main program flow in a non-blocking scheme, allowing main program flow to continue execution [57]
CI	*Continuous integration* is a practice in software development which requires frequent code integration to the shared repository, followed by builds and test execution for early detection of the problems
CMMI	*Capability Maturity Model Integration* [44] is a model for processes improvement and benchmarking
CRUD	*Create, read, update, delete* are basic functions applicable for persistent data
DBA	*Database Administrator* is a role charged with design and maintenance of database
DoS attack	*Denial of Service* is a type of cyber attack where attackers are aiming to reduce accessibility of the attacked service
HTTP	*Hypertext transfer protocol* is application level data transfer protocol according to ISO OSI reference model [49]
ISO/IEC 25010	Software product quality model
KPI	*Key Performance Indicators* are measurable values demonstrating efficiency in achieving business goals
Mutation testing	Method for evaluation test quality. It is done by introducing a change into the system and test cases are evaluated by their ability to detect the change

Pipe-clean test — Test for ensuring validity of performance test scripts [58]. In order to conduct this kind of test, a testing scripts can be run one or few times with a single user

Protobuf — "Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler" [23]

Regression testing — "Testing activity that is performed to provide confidence that changes do not harm the existing of the software" [79]

RESTful API — Web services which follow REST (Representational State Transfer) architecture, proposed by Roy Fielding [7]

SLA — *Service level agreement* is a contract between a service provider and a user which contains description of a service, its quality, rights and responsibilities of parties

System testing — Testing of complete and integrated system

WebSocket — Protocol over TCP which enables two-way communication between a client and a server [25]

X-road — "The data exchange layer for information systems, is a technological and organizational environment enabling a secure Internet-based data exchange between information systems" [8]

Contents

List of Figures

List of Tables

1 Introduction

Performance testing is an important step in quality assurance of large systems. For critical systems it can assist to acquire required certification: performance testing is capable of assuring if the system is meeting requirements for efficiency of ISO/IEC 25010 standard [9]. Performance testing is also the most important step in optimization of resources utilization. Optimal resource utilization has several advantages: it not only decreases price of cloud services [4], but also makes the system less vulnerable to DoS attacks, as the most efficient their targets are the most resource-consuming parts of the system.

The system under test is a distributed asynchronous system, which enables real-time interaction between agents of different types. Interactions include large number of financial transactions involving external system. The system uses Oracle database. It is also not built with RESTful API constraints, foremost in the sense that the server does store context between requests: this is required by the nature of services provided by the system. The system is built using Java, which has been the most popular programming language since 2001 with only few exceptions [78]. Java is often used for building enterprise systems serving large number of customers. Taking that into consideration, not only this work is important for specific product, but it also creates experience valuable for the industry as a whole.

The system under test is commercial and disclosing sensitive information should be avoided. Therefore, many details like architecture of the system under test, domain-specific aspects and definite results of performance testing are omitted in this work. Instead, the principles of decision-making and general technical details used in this work are in the focus, and this makes it possible to reproduce the result within similar constraints.

The work is not done by the author alone, but as a team project instead. All steps which are mentioned in this work and which were done by other people are marked explicitly.

1.1 Problem Background

Performance testing is often a complex, non-trivial, costly, and sometimes ignored procedure. Whereas SLA usually has complete list of functional requirements, it is quite common that non-functional requirements for performance are not defined at all. As a result, system performance is not tested and performance issues are discovered only in production, which leads to users' dissatisfaction, monetary losses, and increased costs of fixing. Sometimes even known performance issues remain unfixed. In severe cases performance issues lead to system outages [2, 46].

Difficulty of performance testing depends on complexity of the system. Testing WordPress websites is a relatively simple task, which could be done using online tools such as Load Impact [3]. Performance testing of web-applications is more complex, as it requires necessary data stored in a database. Performance of such testing usually presumes CRUD operations done by some number of authenticated users. Classical tools for performance testing of web-applications include Apache JMeter [10] and Gatling [6]. Usually optimization of access to database is the major factor of performance improvement of such systems.

Performance testing of distributed systems becomes more costly as it requires more resources for building test environment. Finally, stateful and asynchronous natures of the system can make its performance testing more complex. In such systems concurrent issues, like livelocks, deadlocks, race conditions, and others may come to the scene in stochastic manner, which are able to make behavior of the system completely unpredictable. All in all, the more complex and large is the system, the more costly performance testing is, and the larger significance it has.

1.2 Goal Setting

The purpose of this work is to create a solution for performance testing of fully integrated system in order to detect performance regressions between releases. From the perspective of the author of current work it includes preparation of load generator, introducing collection of its metrics, and implementation of test scenarios. Preparation of test environment

and monitoring system is responsibility of other team members. Not all, but only key use-cases must be covered in the scope of this work.

Main success criteria is reliability of result provided by tests: if the test passes, then no issues in production should be encountered. It is also required to find the way to minimize the cost of maintenance of designed performance tests. The tests should be easy to run and provide results in convenient way for analysis: it should be easy to find cause of possible failure. The solution should also be capable of conducting all major types of performance tests.

1.3 Related Works

Ian Molyneaux in his book *The Art of Application Performance Testing* [57] described fundamentals of performance testing: what need to be done prior to performance testing, types of performance testing, available tools, how to design proper environment, how to make results trustworthy, which metrics and KPIs to collect, and how to interpret results. This is the main source of information for this work. Kinds and possible processes of performance testing were also described in *Performance Testing Guidance for Web Applications* published by Microsoft Press [54]. The book includes such information like how to integrate performance testing into agile life cycle and how to make the process of performance testing compliant with CMMI [44].

The field of performance testing has been also studied previously in Tallinn University of Technology. In 2016 Kaarel Purde developed approach for performance testing of a web-application in his Bachelor's thesis [75]. The system used HTTP protocol and Apache JMeter was used for load injection. Solution also contained a stub for external system (X-road service). In the same year Viktor Reinok defended his Master's thesis on performance of Java applications [76]. Custom load injector was implemented. Such metrics as response time, number of database queries, and context tree depth were collected. Integration with CI made the solution fully automated. Both theses describe performance testing of a system of much smaller scale than the system under test in current work.

Using real-life experience is very important for this work. For this reason case studies

published by Web Performance deserve attention [40]. This is an outsourcing company which specialize on functional and performance testing of web services, and it has shared its experience. Case studies do not describe how the testing solutions were designed, however, these works have good examples of performance requirements and test reports, as well as description of discovered performance issues and how they were handled.

1.4 Outline of the Thesis

Section 2 of the thesis starts with description of types of performance tests the developed solution is aimed to support. Section 2.2 describes process of performance testing, which was followed during current work. Section 2.3 has brief description of toolset used during this project which is also usable for performance testing of other large-scale Java systems.

The goal of Section 3 is to illustrate how the proposed process was followed and how required solution was implemented. It discloses implementation of system or pipe-clean and performance tests. The section also demonstrates results of pipe-clean testing. However, results of performance tests are not included due to confidentiality reasons.

Finally, Section 4 provides comparison with other works mentioned in Section 1.3, evaluation of gained results, and explanation of the logic behind decisions made during this work. Advantages and drawbacks of designed solution are included, as well as description of other possible options. At the end of the section things for future development are listed and described.

2 Methodology

2.1 Types of Performance Tests

The goal of designed solution is to enable team to conduct three major types of performance tests: load, stress, and endurance tests. Table 1 contains a brief description of these types of tests.

Test type	Definition
Load test	Test non-functional requirements with the load anticipated in production
Stress test	Find at which load and how the system fails
Stability test	Long-running test for detection of problems like memory leaks

Table 1: Main types of performance tests

During *load test* the system under test is exposed to the load anticipated in production. In other words, load tests are the closest approximation to the real-life application usage. Performance requirements (availability, number of concurrent users, average or maximum response time, etc) are tested with load tests.

Stress test is a category of performance test which aim is to expose the system under test higher load than anticipated in production. Stress test provides such information as capacity limits of the system, at which point and how does the system or one of its components fail.

The objective of *endurance test* is to detect problems which may unveil themselves only after extended time period, for example, memory leaks.

These are the main types of performance tests. Some authors distinguish other types as well, for example, *spike tests* [53]. In context of current work the term *regression test* should be mentioned as this is the major goal of the project. As a separate procedure conducting maintenance works on the system being under load in testing environment

could be also mentioned: it not only assures viability of the system, but is also a useful training for personnel. In practice, it is also possible to conduct isolated performance tests for limited parts of the system. Conducting such tests is not the goal of the current work: instead, the complete system is the subject of performance tests designed for this work.

2.2 Process of Performance Testing

The process proposed by Ian Molyneaux has been taken as the basis for current work [59]. Before proceeding to proposed steps, it is required to assure that the system meets functional requirements. Other important thing is to allocate sufficient amount of resources for performance testing: limited resources is the major reason why performance testing is not done at all. Steps of the proposed process are explained briefly in Table 2.

	Step	Description
1	Non-functional requirements capture	Testable Include normal and peak loads
2	Performance test environment build	Close replica of production environment Similar to production database volume Emulators of external systems Consider load balancing strategy
3	Use-case scripting	Behavior of users Wait time considered Usually version-dependent
4	Performance test scenario build	Duration of the test Number of concurrent users Load injection profiling
5	Performance test execution	Run pipe-clean tests before Test environment should not be used by others
6	Post-test analysis and reporting	Use multiple statistics Visual data representation Log files, heap and thread dumps

Table 2: Main steps in performance testing

2.2.1 Non-Functional Requirements Capture

Performance requirements are mandatory features of the system: failing to meet requirements leads to delay of release. They could be obtained from SLA, significant stakeholder [55], or from observations of functioning system in case of regression performance testing. As any requirements, they should be testable. Here are a couple examples of good requirements:

When N users login within one minute, 90% of all login requests should receive response

within X milliseconds.

When N users are executing use-case scenario X concurrently, then CPU utilization of service Y should not exceed 70%.

Performance requirements should consider not only average load, but also peak load. For example, usage of study information system is relatively low during most of the time, but at certain periods of academic year, such as exam session, the number of concurrent users increases significantly [60].

2.2.2 Performance Test Environment Build

Environment for performance testing should be as close replica of production environment as possible. Only in this case results of performance tests are reliable. Practice shows, that misconfiguration is a common reason of performance issues [47, 48], and it is only possible to find such errors if testing and production environments are identical. The problem is that replicating production environment can be costly for organization.

External systems could be replaced with emulators, like in the Bachelor's thesis of Kaarel Purde [75]. One should remember, that the external system itself might be the bottleneck for performance. Furthermore, one should check if load balancing strategy [61] does not prevent from conduction of performance tests, and that volumes of databases in testing and production environments are similar.

2.2.3 Use-case Scripting

Use-case is a process done by user to achieve his or her goal. It usually includes logging in, making some actions, and logging out. Wait time should be considered as well, as human beings do not act in fractions of seconds. This wait time should not affect measuring of response time.

Use-case scripts are usually version-dependent: new release is potentially able to partially or completely invalidate this scripts [62]. This is the major concern from performance tests' maintenance point of view.

2.2.4 Performance Test Scenario Build

Performance test scenario [63] defines such parameters of test as number of concurrent users and duration of the test. These are the key features which define type of performance test being executed. In addition, test scenario is characterized by load injection profile, or how users are connected. If the system serves *n* concurrent users, it does not mean that all of them are logged in at the same moment. In this case load to the system may be too high. For this reason connection of virtual users should be spread over time.

2.2.5 Performance Test Execution

Performance test execution is the most straightforward step. Ideally, it should be solely a validation of performance targets, but not a bug-fixing activity. Before running performance tests, the team should execute pipe-clean test to check validity of use-case scripts.

2.2.6 Post-Test Analysis and Reporting

Final step is analysis of test results and reporting [56, 64]. Report must provide clear answer whether the system meets its performance requirements and provide clear starting point for root cause analysis of detected problems. Visual data representation with graphs is an advantage.

2.2.7 Other Steps

The process taken as a basis does not presume usage of a custom solution. As this is the case in current work, a few details should be done: system for collection of metrics and load generator.

For system performance analysis metrics like utilization of resources (CPU, RAM, etc) of each component of the system under test, response time, request rate, and number of concurrent users should be collected. When it comes to response time, only mean or maximum response time is not enough. For example, there might be 10-fold difference

between 90th percentile and maximum response time.

Load generator is a common bottleneck of the performance testing process. For this reason it should be thoroughly monitored as well. For large systems multiple load generators might be required.

2.3 Used Tools and Technologies

There are number of tools and technologies for monitoring JVM applications which can help during development, ranging from standard VisualVM [74] to commercial XRebel [81]. Thesis of Viktor Reinok has overview of these tools [76]. For current work VisualVM was used during optimization of load generator.

Zabbix [52] has been used for monitoring of utilization of resources of each component of the system. For collecting metrics from load generator Spring Boot Actuator [37] was used. Actuator enables endpoints, which provide information about Spring Boot [36] Application. */metrics* endpoint provides application's performance metrics, like amount of total and free memory, uptime, heap size, etc. Actuator also supports custom metrics: gauges for single values, counters for measuring requests throughput, and timers for measuring latency. These options were used for collecting metrics on response times and error rates. Accessing */metrics* endpoint only shows metrics at specific point in time. However, for performance testing metrics values throughout the whole time period of test are required.

Since version 2 of Spring Boot the Actuator relies on Micrometer [32], which is a facade for instrumentation framework, for example, Prometheus [17], which was used in this work. Prometheus makes regular requests to Actuator's endpoints, fetches metrics, and stores them in a time-series database. It enables to see values of collected metrics at any point of time during observation. Even though acquired data can be displayed in Prometheus itself, Grafana [50] is considered as a more decent tool for data visualization. PromQL, or *Prometheus Query Language* has been used for building graphs [19].

Log analysis was done using Graylog [26]. It is a log aggregator which enables to analyze logs from different machines in a single place.

As the solution is built using Java, it is important to remember about changes in licensing of Oracle Java SE since 2019 and prefer OpenJDK over Oracle JDK [72].

3 Implementation and Results

System and performance tests were implemented using existing component called *bot-server*. Before proceeding to implementation of tests themselves, the architecture of bot-server is explained.

3.1 Bot-Server as a Load Generator

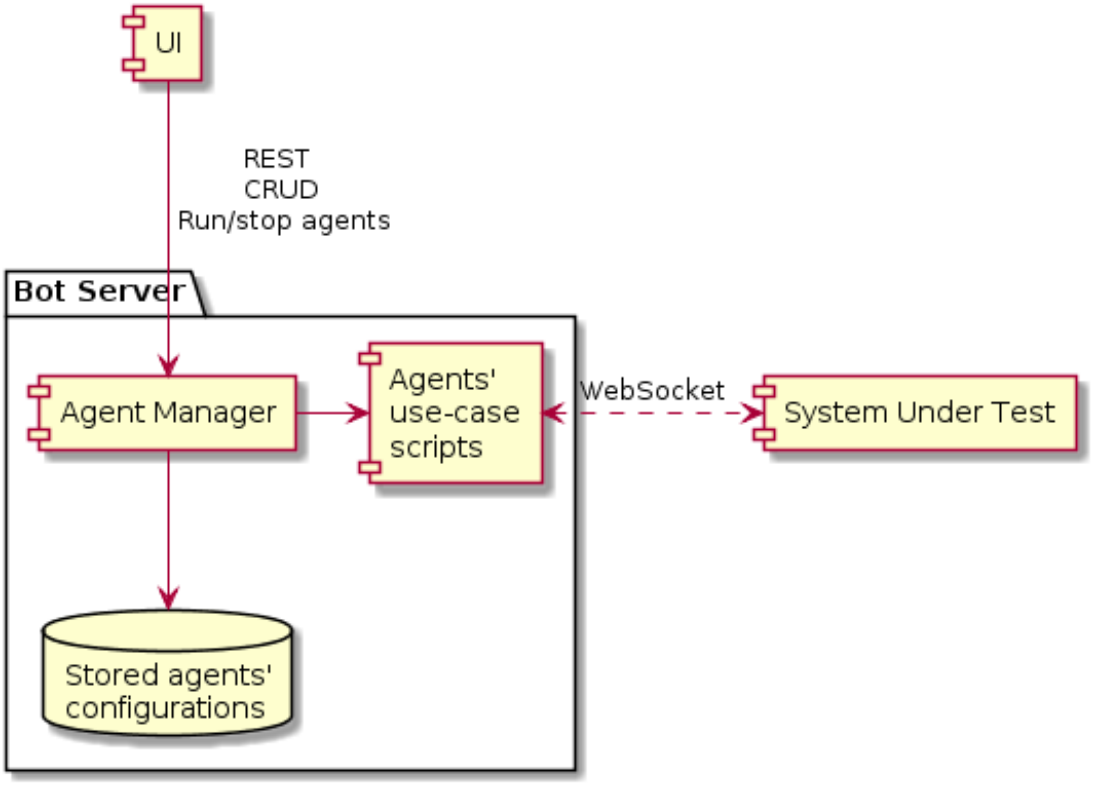

Figure 1: Architecture of bot-server

Bot-server is a system's component designed for managing automated agents which are using the system under test. They serve for number of purposes, and they are able to interact with other agents or with human beings. When idea of using bot-server for performance tests emerged, it was accepted by the team.

There are two types of agents in the system. Relatively small number of agents of type 1

interact with large number of agents of type 2. For this reason type 1 agents are launched individually, and type 2 agents are launched in batches of up to 100 virtual users having the same configuration. Automated agents of type 2 are configurable to run specific use-cases. They also generate the largest part of load and transactions in the system.

While managing of automated agents is done with CRUD operations via RESTful API, agents themselves communicate with the rest of the system via WebSocket [25]. There is no communication between agents inside bot-server: all interactions are conducted via the system under test. Also, communication is only possible between agents of different types.

Asynchronous steps and requests in bot-server are implemented using Project Reactor [34] and Google Guava's *ListenableFuture* [21].

Initially the author of current work was one of ordinary contributors of the bot-server, who implemented bots for few use-cases and fixed some defects. The architecture of the bot-server was not designed by the author. However, as a result of current project the author became one of the key maintainers of this component, who detects and fixes issues in use-case scenarios and introduces new features.

3.2 Required Changes in Bot-Server

Bot-server has not been originally designed for system or performance tests. This fact required introduction of few changes. Both system and performance tests presume processing of incoming and outgoing requests: system tests for validation (some messages should have expected values in specific fields, and other messages should not be sent or received at all), and performance tests for collecting metrics, such as rate of requests or response time. *Observer* design pattern [20] was used for acquiring requests and responses from the automated agents; latter processing depended on type of test.

Use of Micrometer required migration of bot-server to Spring Boot.

3.3 Functional Tesing: System and Pipe-Clean Tests

Designed functional tests serve two purposes. Their major goal was system testing, and they were also used for validation of use-case scenarios as discussed in Section 2.2.5.

There are 18 major use-case scenarios in the system, and the bot-server currently supports 12 of them. Architecture of the service allows to cover all of them with only one test scenario script, which takes use-case as a parameter.

3.3.1 Implementation

The main idea of design of functional tests is based on running limited number of automated agents with strict validation of the most important incoming and outgoing requests. Designed tests are triggered from Jenkins [45] with multiple parameters: duration, environment to run in, number of virtual users, use-case scenario, and git branch.

The test was implemented using JUnit 5 [77]. Automated agents are launched in the main thread, which is then paused by calling *Object.wait(timeout)* [69] on specific object. This shared object is accessible from all threads and it notifies main thread in case of error, and it also has a string field for storing details about error. After launching, automated agents act asynchronously. Each automated agent has its own observer, were validation of requests and responses occurs and which contains reference to shared object. In case of any unexpected request or response, the error message is set to shared object and the signal is passed to the main thread with *Object.notify()* [68]. Spurious invocations of *Object.notify()* are handled by checking failure condition in *while* loop, as recommended by Oracle [69]. Attributes of shared object are stored as atomic [73] variables, so that the updated values are seen between threads. Described test flow is also demonstrated on Figure 2.

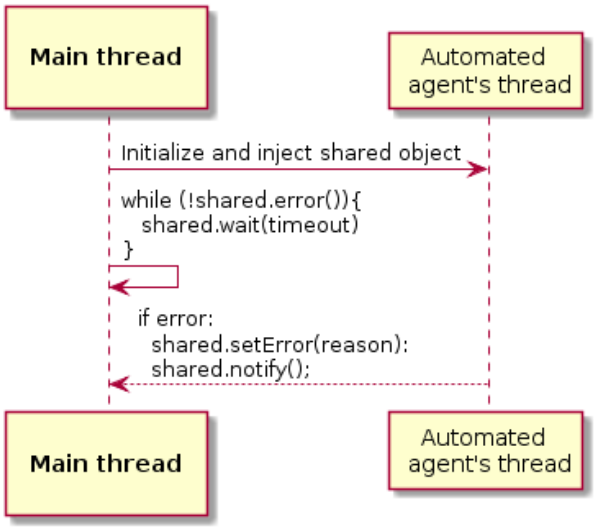

Figure 2: Communication between threads in functional tests

3.3.2 Results

As Table 3 shows, distribution of found errors was unequal between two tested components: much more errors were found in load generator's use-case scripts than in the system under test. As mentioned in Section 2.2.3, the reason was version-dependence of use-case scenarios.

Two errors in the system under test were found, and one of them had large significance.

Functional tests had quite large number of false positive results. This was caused by usage of shared environment. For example, some number of test failures were caused by the system's redeployment by other team members during the test. For this reason at some stage both functional and non-functional testing has been completely moved to dedicated environment.

Number of executions	85
Number of errors found in use-case scenarios	15
Number of errors found in the system under test	2
Number of false positives	12

Table 3: Results of system and pipe-clean tests

3.4 Non-Functional Testing: Performance Tests

This section describes how the solution for performance testing was implemented and how performance testing is conducted, based on the process listed in Section 2.2. The section does not describe how test environment was built, as it was done by other people. This section also does not describe how use-case scenarios were implemented: existing use-case scripts were reused and new ones were not created during this project; instead, existing use-case scripts were fixed as a result of functional testing.

3.4.1 Non-Functional Requirements Capture

Main objective of designed performance tests is detection of possible performance regressions. This means that major source of requirements is live operations team. For confidentiality reasons collected requirements can not be listed in current work, however, it is possible to provide an insight what do these requirements include. First thing was to define scenarios which cause the highest load of the system in production. Scenarios include name of business process, duration of the process, and number of concurrent users. Names of interviewed persons were also included. Most critical requests and their error rates in production have been considered as well.

Other important source of requirements is development team implementing architectural changes. Processes, which were the most affected by architectural changes, were collected. QA team provided detailed functional requirements. Deadlines were set by the head of department.

3.4.2 Generation of Test Data

The database in testing environment has been populated by making copy from production database with obfuscation of sensitive information, like usernames. This part has not been done by the author of current work.

There was also need in database entries which are referenced in test scripts. At early stages of the project required objects were created manually. However, the script for generating

INSERT SQL queries has been written later for automatic creation of any number of required objects. The script was written for only one required use-case out of 12, and the selection of the use-case was based on its importance in real life. Generated query created objects with negative ID's in order to avoid collisions with ID's generated by database sequences.

3.4.3 Implementation of Test Scenarios

Test scenarios are created with a web form by a tester before execution. Following fields have to be filled: use-case, number of agents of type 1, number of agents of type 2, number of iterations, and also time interval in seconds within which all virtual users are logged in and start activity. In case multiple use-case scenarios need to be launched in parallel, which is closer to real-life conditions, then the form is filled again.

3.4.4 Load Generator

Load generation is currently done with only one server. When it works at full capacity, all crucial services in a cluster have from 80% to 90% of CPU utilization (Figure 15). All services still remain responsive. At the same time, CPU utilization of the load generator itself is approaching 100%, as demonstrated on Figure 3. Memory utilization is not a bottleneck of the load generator, as shown on Figure 4 (green correspnds to eden space, orange: old generation space, and blue: survivor space).

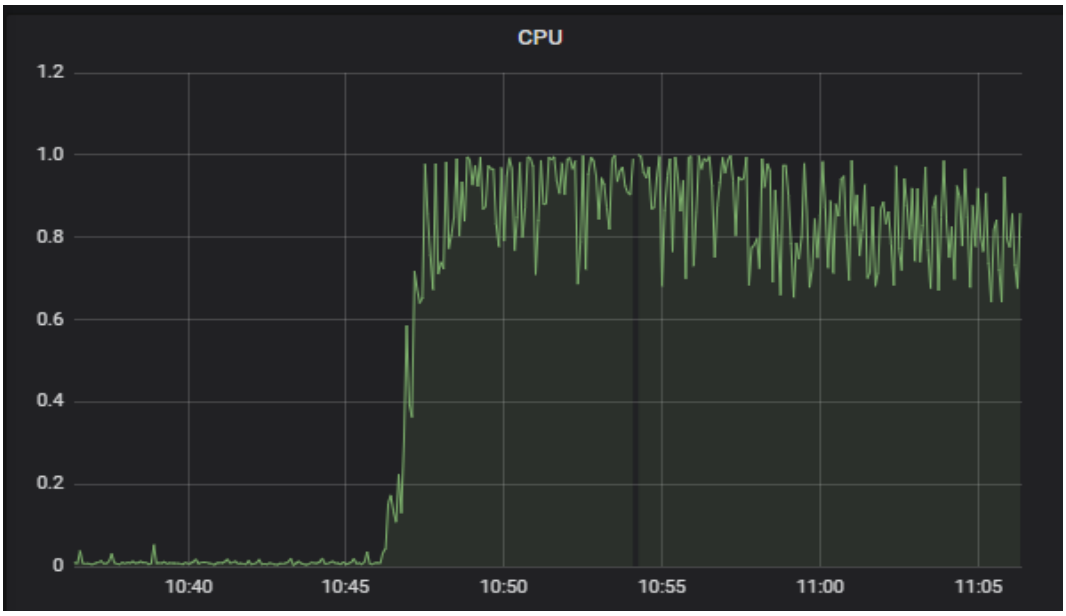

Figure 3: CPU utilization of the load generator at the highest load

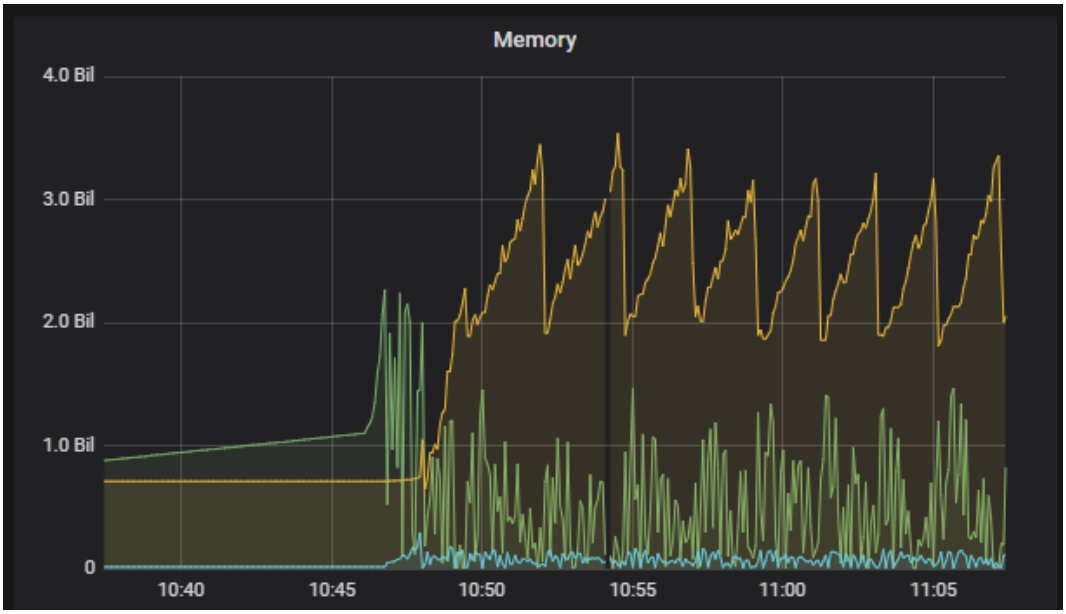

Figure 4: Memory utilization of the load generator at the highest load

3.4.5 Collection of Metrics

Spring Boot Actuator with Micrometer was chosen for collection of metrics as a standard solution for Spring Boot Applications. It is thread safe, which is very important

for load generator as a concurrent environment. Request times were collected using *Timer.Sample* [31], throughput of requests was collected with *Counter* [28], and number of concurrent virtual users was measured with *Gauge* [29, 30].

	System under test	Load Generator	Tool
Utilization of resources (CPU, RAM, etc)	+	+	Zabbix
Number of threads		+	VisualVM, Spring Boot Admin
Number of concurrent users		+	Micrometer, Grafana
Requests per minute		+	Micrometer, Grafana
Response time		+	Micrometer, Grafana

Table 4: Metrics collected by designed solution

Table 4 demonstrates main metrics collected by designed solution, which were observed during trial performance tests and optimization of the load generator. There are much more metrics related to resource utilization collected by Zabbix, however, these are the starting point for analysis of performance tests' results.

The author of current work was responsible for metrics collection from load generator. Figures below demonstrate, how collection of metrics was introduced. Examples of graphs produced by these samples are shown in Section 3.4.7.

For measuring requests rate per unit of time, the *Counter* [28] from Micrometer framework must first be introduced. Its usage is demonstrated on Figure 5.

```
private MeterRegistry meterRegistry;

public void onUserActionRequest() {
  meterRegistry.counter("user.action.request").increment();
}
```

Figure 5: Usage of Micrometer's Counter

Figure 5 also demonstrates correct usage of naming conventions [22].

In order to display result in Grafana, PromQL's *rate()* [18] function was used, which is shown on Figure 6.

```
rate(user_action_request[1m])
```

Figure 6: PromQL's *rate()* function

In case of responses it was crucial to measure not only their rates, but also distribution by response statuses. Response in this context means not HTTP status, but the logical status which indicated result of request processing. This was achieved by using tags, like shown on Figure 7.

```
public void onUserActionResponse(Response response) {
  meterRegistry.counter("user.action.response", "status", response.
      getStatus().name()).increment();
}
```

Figure 7: Usage of Micrometer's Counter with Tags

It is also important to count number of active users at each point of time during performance testing. *Gauge* [30, 29] is the right class to achieve this. Figure 8 demonstrates how it was used.

```
@PostConstruct
public void countActiveUsers() {
  Gauge.builder("users.active.count", this::activeUsersCount)
      .register(meterRegistry);
}
```

Figure 8: Usage of Micrometer's Gauge

Finally, it is important to measure response times of the most critical queries. It was done using *Timer* [33, 31], which is demonstrated on Figure 9.

```
private Timer.Sample userActionSample = buildTimer("user.action.
    response-time");

private Timer buildTimer(String name) {
  return Timer.builder(name)
      .publishPercentiles(0.1, 0.9)
      .publishPercentileHistogram()
      .distributionStatisticExpiry(Duration.ofMinutes(30))
      .minimumExpectedValue(Duration.ofMillis(1))
      .maximumExpectedValue(Duration.ofSeconds(10))
      .register(meterRegistry);
}

public void onUserActionnRequest() {
  meterRegistry.counter("user.action.request").increment();
  userActionSample = Timer.start();
}

public void onUserActionResponse(Response response) {
  userActionSample.stop(loginTimer);
  meterRegistry.counter("user.action.response", "status", response.
      getStatus().name()).increment();
}
```

Figure 9: Usage of Micrometer's Timer

Figure 9 also demonstrates how publishing of percentiles was implemented.

3.4.6 Performance Test Execution

Checklist on Figure 10 enumerates all steps needed for execution of designed performance tests.

1. Check that environment is not being used by other team members
2. Check that deployed version of the system is stable and passes unit and integration tests and has no functional errors
3. Deploy required version of the system and load generator to testing environment
4. Check in Spring Boot Admin that all services are up and running
5. Check that system monitoring (Zabbix and Grafana) works properly
6. Check that Graylog receives logs from all services
7. Check that objects referenced in use-case scripts are stored in database
8. Execute pipe-clean tests for validation of use-case scenarios
9. Create testing scenario with web form and start the test

Figure 10: Checklist for performance test execution

Apparently, most of the steps are just checks. If they are done, then the team would not encounter such cases like discovering that logging did not work correctly during testing after long-running test, or struggling to detect issues only to find that the use-case scenarios was non-valid.

3.4.7 Post-Test Analysis

This section describes the process facilitated by designed solution for performance testing. Actual results are not added here for confidentiality reasons. However, some sample graphs acquired by designed solution are present.

First thing to analyze in results of performance testing is number of concurrent users and its change over the time. All other metrics should correlate with this graph. Each case of decreased number of users may indicate some sort of failure caused by performance, concurrency, or functional issue. An example is demonstrated on Figure 11.

Figure 11: Monitoring of number of concurrent virtual users

Next thing to analyze is a rate of request and responses during performance test. Figure 12 demonstrates how rates of clients' incoming messages are monitored.

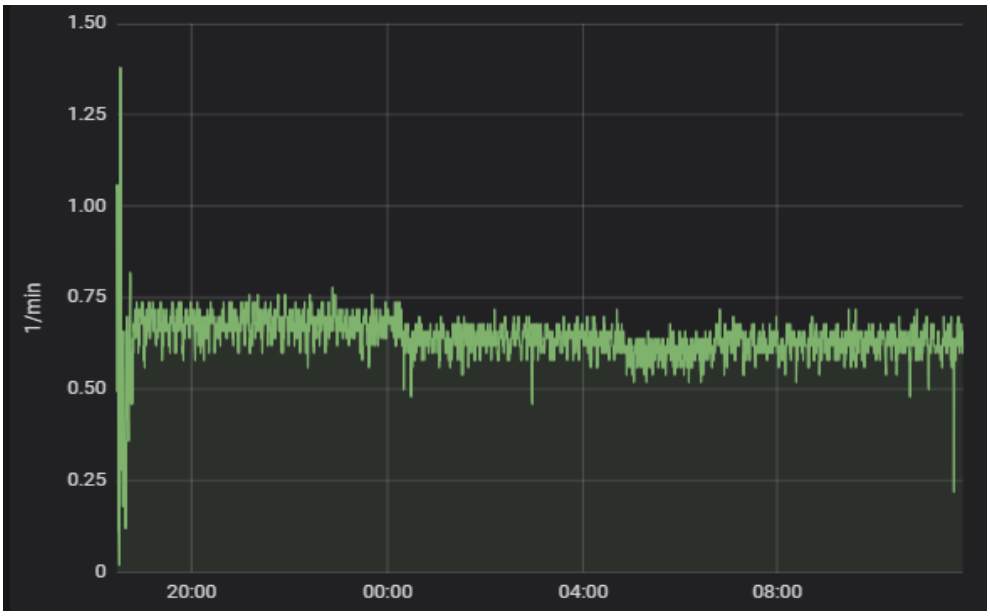

Figure 12: Monitoring of request rate

When it comes to requests sent by clients, not only request rate is important, but also rate of responses with distribution by response status. This helps to check if permitted by requirements error rate was exceeded or not. Upper green graphs on Figure 13 shows rate of requests per minute, blue represents successful responses, and other colors stand for responses with errors of different kinds.

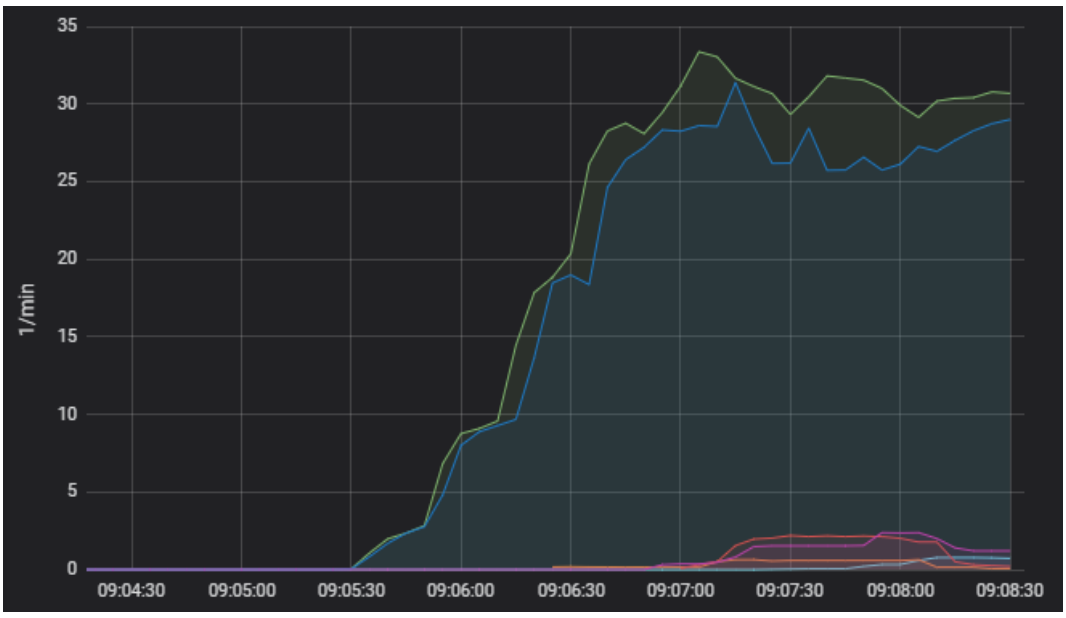

Figure 13: Monitoring of request rate with tags

Figure 14 demonstrates how monitoring of response time was implemented. Currently, it shows average, maximum response times, and also required percentiles. Percentiles to be shown are defined by use of Micrometer's Timer, which is shown on Figure 9.

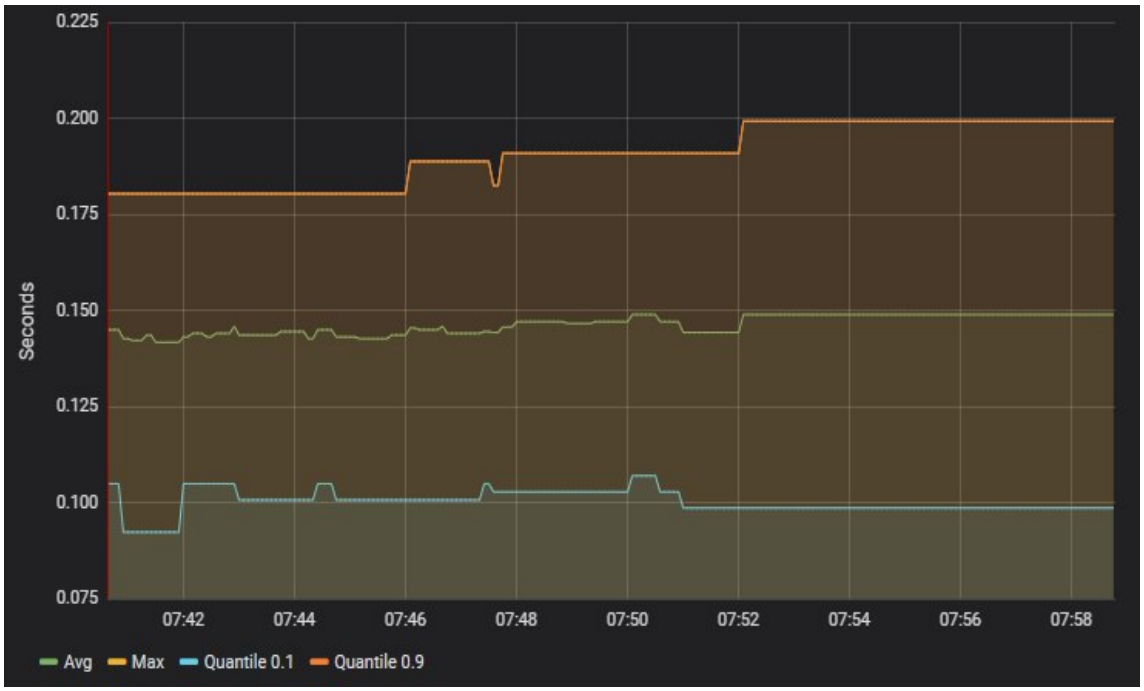

Figure 14: Monitoring of response time

Then, utilization of resources is analyzed in Zabbix (Figure 15, Figure 16). This should be checked for all components of the system under test including load generator.

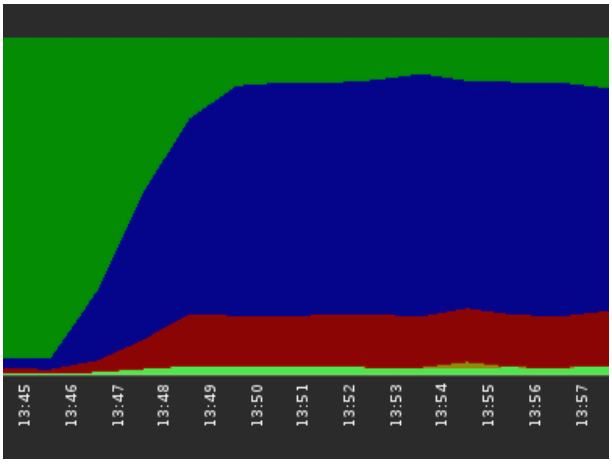

Figure 15: Utilization of CPU by one of the system's components

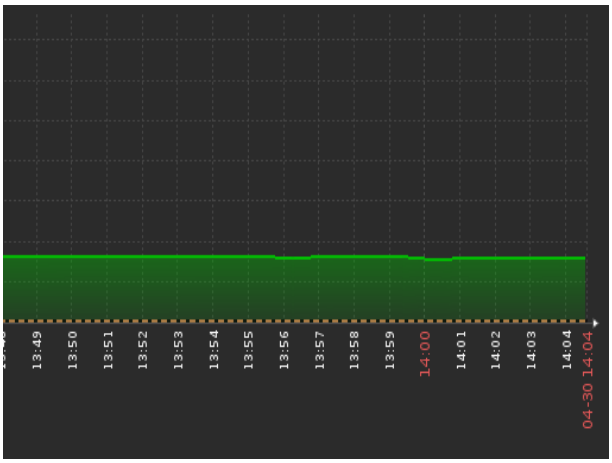

Figure 16: Utilization of memory by one of the system's components

If the test has been conducted for sufficient period, then RAM utilization could be checked for presence of any memory leak.

Another thing to check is whether the load is distributed equally between all servers in the cluster. If not, this can indicate some sort of error in test scenario configuration, or even in a load balancer.

Final step is log analysis, which is done using Graylog. One should select proper day interval with UI date picker, and then insert search query [27] shown on Figure 17.

```
environment:performance_testing AND level:<6
```

Figure 17: Graylog query for acquiring errors and warning generated during performance testing

Collected metrics are compared to values defined in non-functional requirements. Analysis step also includes comparison of current and previous test results. In current solution results of previous executions could be found in the same monitoring systems.

As a result of analysis it must be clear whether the sysem meets non-functional requirements and are there any performance regressions.

4 Analysis

4.1 Comparison with Other Solutions

4.1.1 Apache JMeter

This work would be complete without any comparison between designed solution and the top tool for performance testing: Apache JMeter. It was used in Bachelor thesis of Kaarel Purde [75]. Only response time of the system under test was analyzed in that work.

Apache JMeter is a standard tool, meaning support of community and its knowledge on labor market. It is also extremely flexible. Initially it was designed for testing of web-applications, however, today it is suitable for testing of more complex systems. Apache JMeter supports multi-server load generators, where one instance is a master, and others are slaves [14], which could be useful for current project. WebSocket support can be added with plugins [80], as well as monitoring of utilization of resources [15]. Apache JMeter provides reporting out of the box, however, if the team prefers Grafana, it is possible to write test data to InfluxDB [1] and then display it with Grafana [13].

However, JMeter does has no support of Protobuf [23], used for serialization of messages in the system. It is still possible to write test plans in Java rather than in .JMX format in order to overcome this obstacle. It is doubtful though that such solution would be easier to implement than a custom one, or bring any advantages. Moreover, it is also an advantage if same monitoring system is used in production and testing environments. For these reasons Apache JMeter has not been used for this project.

It is worth to mention that designed during this work solution supports similar definition of test properties with UI, as Apache JMeter does, where properties are *Number of Threads (users)*, *Ramp-Up Period (in seconds)*, and *Loop Count* [11, 12], which is demonstrated on Figure 18:

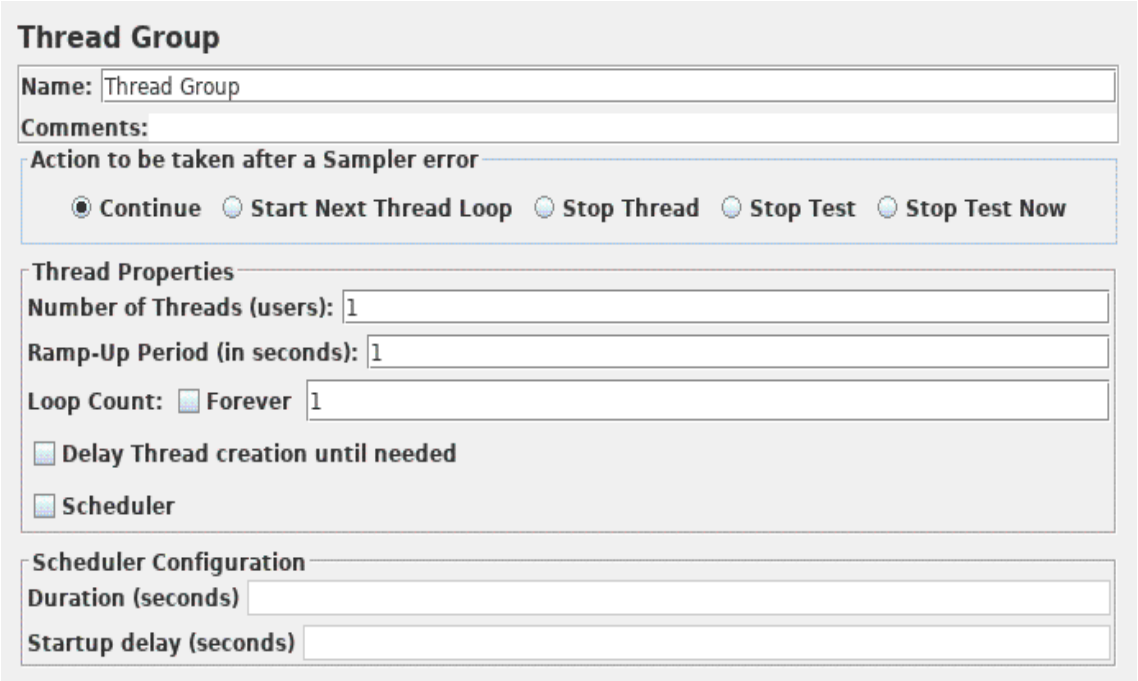

Figure 18: Apache JMeter, UI for definition of test scenario [11]

4.1.2 Solution of Viktor Reinok

In 2016 in his Master's thesis Viktor Reinok proposed solution for performance evaluation of Java applications [76]. Not only HTTP request time is measured, but also program's context tree size and number of database queries per each request. Proposed solution also compares acquired test result with previous one automatically. Moreover, the solution is fully automated and can be integrated in CI. However, it is not clear whether the solution differentiates between fetching data from database or from cache, which is implemented in many projects using Spring [39], Hibernate [24] or Redis [51].

Utilization of resources is not measured in proposed solution. It was also designed for a monolith application, which is not the case for current project. For this reason in context of current work, such solution could be useful for isolated performance testing of some services of the system.

4.1.3 Web Performance Load Tester

Web Performance conducts testing of less complex systems than one tested during this work. The company is focused on testing of web-applications. Nevertheless, their case studies [40] still deserve attention, because the company produced high-quality performance testing tool for systems which should be able to handle thousands or even millions of concurrent users, like a state-level election website [41]. The main question is whether the developed solution is suitable for same-level performance analysis. Investigation in provided case studies rely on the same metrics collected with current solution, so the answer is yes: the designed solution is suitable for performance analysis of enterprise scale systems.

It is important to note that provided case studies demonstrate that competence of performance testers play crucial role in analysis: skills of DBA, live operations, and system administration teams are required for detailed analysis. Also, the case studies demonstrate that quite often the configuration is the reason of performance deficiencies [42], and this proves why making testing environment a full copy of production environment is so important.

Case studies stress the importance of database monitoring [42, 43]. This is the main thing to improve in designed solution and process.

4.1.4 Previous Solution for Performance Testing

In the beginning of the project there were two options for load generation. First was to use existing solution for performance testing. It has already been used for few years. Moreover, the solution had simple implementation. At the same time the team was looking for the way to diminish maintenance cost of performance testing solution. This is how the idea of using bot-server appeared: in addition to its direct tasks, bot-server could be also reused for load generation in performance testing. Being used in production means thorough detection of errors and their fixing with high priority. Disadvantage of this approach is its relative complexity compared to the first option. This disadvantage was partially diminished by implementation of functional tests for use-case scripts and test

scenarios in new solution.

Besides simplicity and maintenance, two solutions have other differences. Firstly, virtual users in the original solution had synchronous behavior: for example, all virtual users are logged in at the same time, and when the login of the last one is done, then agents proceed to the next step all together. On the contrary, in bot-server agents act asynchronously at their own pace. This behavior is closer to real life.

Secondly, new solution provides wider range of metrics: while old solution collects and analyzes only server response times, new solution also provides information about utilization of resources by the system under test and load generator itself.

Both solutions highly depend on data entries, which should be stored in database prior to starting of test. This work includes automated solution for generation of SQL scripts for inserting this data.

Finally, while execution of newly developed performance tests is easy, it is still manual, unlike previous solution, which had fully automated and scheduled test jobs.

4.2 Consequences of Decision of Load Generation

Intensive usage of bot-server in system and performance tests meant thorough testing of bot-server itself. Shortly before start of this project the bot-server underwent fundamental changes: its functionality was significantly enhanced. In addition, changes in communication protocol were made. Cost of such dramatic changes was introduction of new bugs. Errors in bot-server, not in the system under test, were the first issues detected by designed functional tests.

Decision to migrate bot-server to Spring Boot for enabling usage of Micrometer for collection of metrics had positive impact. Deployment of bot-server became easier. Furthermore, it enabled to use Spring Boot Admin [38] for management of the service.

4.3 Possible Design Patterns for Processing of Messages

Invocation of methods for collection of metrics could be added directly to the places where they are required. Design of this approach is the simplest, however, it breaks single responsibility principle for classes and methods. Moreover, this approach is highly invasive, as the changes are required in large number of classes. Using *Observer* [20] OOP design pattern and concentrating functionality for collection of metrics facilitates its testing. Since an error in observer may violate main flow of a process, unit-testing is important. Replacing Micrometer with other framework is also easier in this case, even though its necessity is improbable. Final advantage of such approach is an ability of storing state for collecting metrics, which in this case were timers, which started on request, and stopped on responses. It was beneficial in cases when handling of requests and responses is made in different classes. However, observer still needs to be invoked from the points of interest. If intervention into existing code would not be possible, then observers could be added and invoked using Javassist [5]. Same approach was used by Viktor Reinok in his Master's thesis [76].

Other possible approaches are based on using proxies. *InvocationHandler* [66] in Java enables creation of dynamic proxies. This approach requires minimal changes in existing code. Necessity to create relatively large number of new classes is a drawback.

Aspect-oriented programming could be another option. Like previous one, this approach avoids changing existing code and mixing different-purpose functionality in same methods and classes. Its implementation in Java is based on proxies. While original implementation is AspectJ [16], Spring AOP is easier to use [35]. However, the latter can only be applied for beans managed by Spring container, whereas the former is applicable for all classes.

4.4 Functional Testing: System and Pipe-Clean Tests

Execution of system test requires existing entities in database. As the number of objects is small, it is not a problem. Possibility to run tests with different Jenkins parameters is an advantage, which makes these tests flexible. For this reason, there is a possibility of scheduling automated launches of the tests. Another advantage is the fact that all use-

case scripts are able to be tested with the same test scenario due to generic nature of the processes.

The tests though do require some investigations in case of failures, because some errors are caused by the fact that tests run in environments used by other team members.

Currently, majority of errors discovered by designed functional tests are found in load generator's use-case scripts. To be more reliable and trustworthy source of information as system tests, they need some history or working without any failures in load generator.

4.5 Non-Functional Testing: Performance Tests

4.5.1 Generation of Test Data

Making copy of production database is probably the best solution for populating database. This makes results of performance tests sufficiently reliable, as the data has the same volume as in production.

When it comes to data referenced directly in test scripts, then using scripts for automatic generation is highly preferable over manual input. The only advantage of the latter approach is that it always produces valid results. However, database schema does not undergo significant changes in such mature system as the system under test, so the importance of this factor is rather negligible. Automatic generation of test data becomes critical if necessity of resetting database to its initial state after execution of test occurs. Otherwise, time-consuming routine of manual test data input has to be repeated before each launching of performance tests.

At the same time it is beneficial that automatic data generation has not been used for population of database to production volume: in this case production and testing databases would have different distribution of saved entries over the data structure used for storing data.

4.5.2 Implementation of Test Scenarios

Even without introduction of designed form for defining test scenarios, it was still possible to run performance tests: as described in Section 3.1, automated agents could be launched in batches of up to 100 virtual users. However, this approach was less convenient as designed solution and it also was more error prone in terms of launching correct number of virtual users.

Other problem that has been solved is too fast connection of virtual users at the beginning of the test. Too fast starts produced the load higher than expected. Figure 19 demonstrates spike in CPU utilization of load generator.

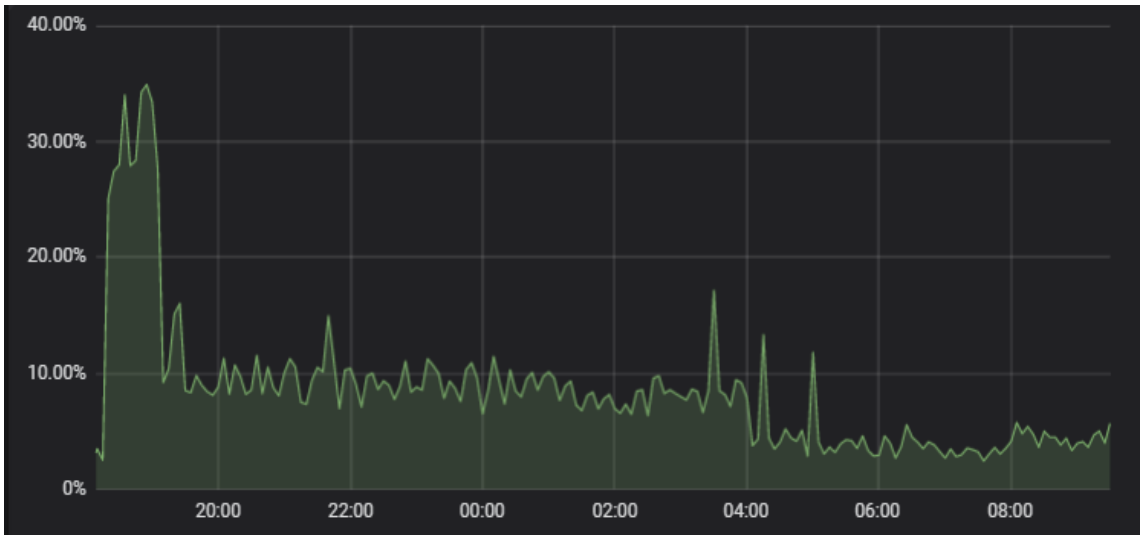

Figure 19: Increased CPU utilization of load generator at performance test start

Number of virtual users was almost the same throughout the test. Such spikes are accompanied by higher error rates in the system:

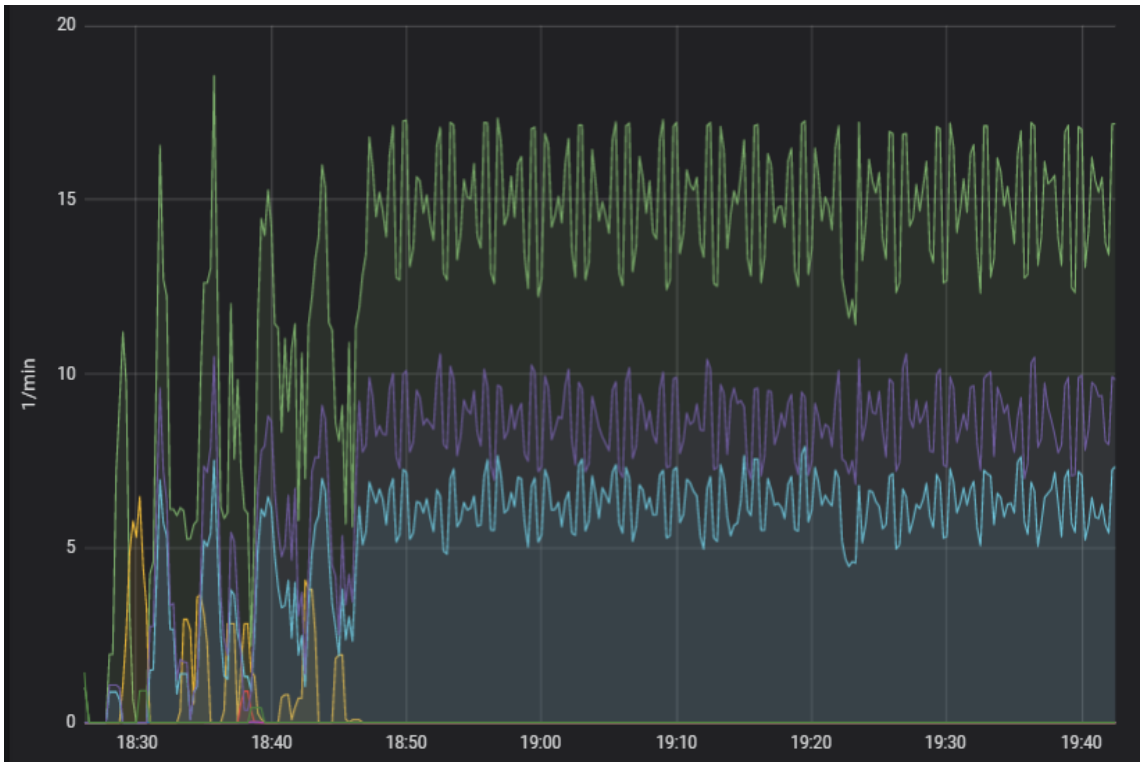

Figure 20: Increased error rate at the start of the test

Green graph on Figure 20 corresponds to requests, purple and blue lines correspond to valid responses. Other lines in the beginning of graph correspond to errors caused by spike of load. Such cases should only be tested if they are indeed anticipated in production. Launching virtual users more smoothly as demonstrated on Figure 21 is more preferable as it enables to avoid such spikes, which was done during this work.

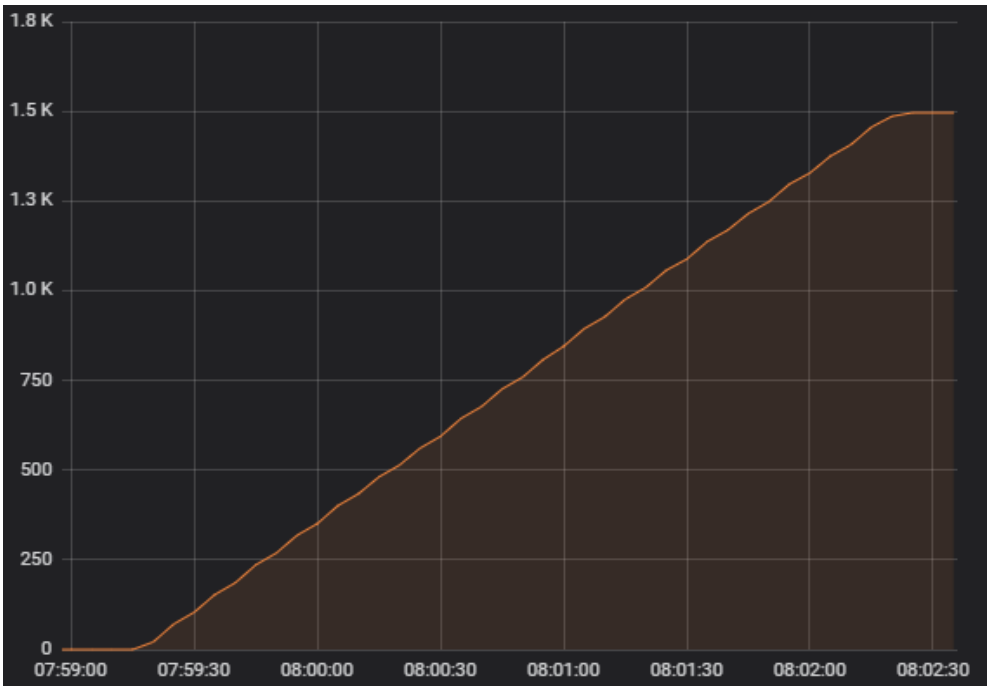

Figure 21: Smooth ramp-up implemented in this work

4.5.3 Performance of Load Generator

In case of Micrometer, *Timers* require special attention as this is the most memory-consuming metric [33]. For this reason response time is measured not for all, but only for the most crucial requests. However, as Figure 4 demonstrates, currently the memory is not a bottleneck of performance of load generator. On earlier stages of development CPU utilization was smaller with the same number of virtual users. For this reason optimization should be tried before adding more capacity or building multi-server load generator.

Taking that into consideration, designed solution is ready for load and endurance tests, but for stress tests it requires more optimization.

4.5.4 Ease of Running

Launching of designed performance tests cannot be automated or scheduled. Manual launching with UI of load generator is still easy.

The most significant bottleneck of the process was adding test data to database for entities referenced from use-case scripts. Taking that into consideration, SQL scripts for test data input should be prepared for all remaining use-case scenarios.

4.5.5 Reporting and Analysis

Current solution provides sufficient number of metrics for performance analysis of the system. Despite the fact that only CPU and RAM monitoring by Zabbix was mentioned so far in this work, the tool provides much wider range of metrics. It is possible to analyze past test executions as well, however, if heap and thread dumps are required, they should be done manually during the test.

Monitoring and reports have things which require improvement. Current solution only makes high-level performance testing. Now it is impossible to detect minor performance regressions due to statistical errors. For example, if performance of the system decreases by 1% with some release, it is impossible to detect that with graphs unless performance meets requirements. It is unlikely though that such minor regressions could be detected with any performance test of fully integrated system at all. For more precise analysis, isolated performance tests for each component are required.

Also, load of a database is currently not analyzed. It is highly desirable as the database is a common bottleneck in system's performance. For database analysis number of executions of each query is required: the most frequent queries should be the first to be optimized. Other thing to investigate is execution plan. Execution time of a query could only be reliable metric only if a database in testing environment has the same volume as in production. Otherwise, even non-optimal queries would require little time for execution. For instance, a query running just 10 ms in development environment can cause severe performance issues in production.

Information on Oracle database statistics can be obtained using dynamic performance views [65], for example, *V$SQL*:

```
SELECT * FROM V$SQL;
```

Figure 22: Query statistics for Oracle database

Using query on Figure 22 information as number of executions, elapsed and CPU time, and cost of specific query can be obtained [70].

Information on database performance can be also obtained in a more convenient way using Oracle Enterprise Manager [71], which is a standard tool for Oracle database monitoring and which relies on using dynamic performance views. Features of Automatic Database Diagnostic Monitor (ADDM) are required for analyzing results of performance testing [67]. ADDM enables to analyze real-time metrics, such as CPU usage, I/O operations, and throughput of transactions. ADDM is also useful for optimization: it provides such information as top SQL statements consuming the largest part of database activity.

4.6 Future Development

Aside from its direct goal, this project had tremendous influence on the design of bot-server. A lot of extra work need to be done though. To start with, plan for refactoring of bot-server has been prepared: as it was developed by different people, in some places it lacks consistency. Besides, fail-fast design approach is going to be implemented in bot-server in order to make detection of errors during exploitation more easily.

In nearest future there is a plan to implement full set of use-case scenarios of the system in load generator, so that tests would be able to provide complete information on system's performance.

Another thing remained undone is export of test reports. While data is can be displayed graphically in monitoring systems in convenient ways, it is not stored there forever. This means that if stakeholders require the report about how the performance of system changed in one year, then this information might not be available until separate storage for acquired data is implemented.

Mutation testing could make evaluation of designed solution more efficient, thus there are plans to conduct this type of test after finishing all features mentioned above.

Finally, it is extremely necessary to optimize CPU utilization of the load generator to make it suitable for stress testing.

5 Summary

The goal of this work was to develop a solution for performance testing for a system, which usage is based on real-time interactions between agents. The solution was based on classical approach for performance testing. The work has special focus on functional testing of use-case scenarios, load generation, and collection of metrics.

For load generation an existing service has been used, and this decision had enormous positive impact on its development. Load generator underwent thorough functional testing, as well as the system under test itself, which enabled to detect large number of issues and fix them with reduced cost.

Designed performance tests are triggered from UI of load generator. Test results can then be observed using Zabbix (utilization of resources of the system under test) and Grafana (number of concurrent users, rate of requests, response time). Functional issues encountered during performance tests are observable as response statuses in Grafana or in Graylog if details are required.

All in all, designed framework is able to provide detailed information about behavior of the system under load and it is already being used for performance testing of real-life enterprise system. Current results seem to be positive, however, the true value of designed solution can only be assessed after months and years of usage. After the completion of this thesis, the development of the solution is going to be continued.

References

[1] InfluxData. https://www.influxdata.com/. Accessed: 2019-04-28.

[2] Post-mortem on the Skype outage. https://beeyeas.blogspot.com/2014/01/cio-update-post-mortem-on-skype-outage.html. Accessed: 2019-04-28.

[3] Load Impact AB. Load Impact. https://loadimpact.com/. Accessed: 2019-04-28.

[4] Amazon. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 2019-04-28.

[5] Shigeru Chiba. Javassist. http://www.javassist.org/. Accessed: 2019-04-28.

[6] Gatling Corp. Gatling. https://gatling.io/. Accessed: 2019-04-28.

[7] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[8] The Nordic Institute for Interoperability Solutions (NIIS). X-Road. https://github.com/nordic-institute/X-Road. Accessed: 2019-04-28.

[9] International Organization for Standardization. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. https://www.iso.org/standard/35733.html. Accessed: 2019-04-28.

[10] The Apache Software Foundation. Apache JMeter. https://jmeter.apache.org/. Accessed: 2019-04-28.

[11] The Apache Software Foundation. Apache JMeter. Building a Web Test Plan. Adding Users. https://jmeter.apache.org/usermanual/build-web-test-plan.html#adding_users. Accessed: 2019-04-28.

[12] The Apache Software Foundation. Apache JMeter. Elements of a Test Plan. Thread Group. https://jmeter.apache.org/usermanual/test_plan.html#thread_group. Accessed: 2019-04-28.

[13] The Apache Software Foundation. Apache JMeter. InfluxDB database configuration. https://jmeter.apache.org/usermanual/realtime-results.html#influxdb_db_configuration. Accessed: 2019-04-28.

[14] The Apache Software Foundation. Apache JMeter. Remote Testing. https://jmeter.apache.org/usermanual/remote-test.html. Accessed: 2019-04-28.

[15] The Apache Software Foundation. Apache JMeter. Servers Performance Monitoring. https://jmeter-plugins.org/wiki/PerfMon/. Accessed: 2019-04-28.

[16] The Eclipse Foundation. The AspectJ Project. https://www.eclipse.org/aspectj/. Accessed: 2019-04-28.

[17] The Linux Foundation. Prometheus. https://prometheus.io/. Accessed: 2019-04-28.

[18] The Linux Foundation. Prometheus Query Functions. https://prometheus.io/docs/prometheus/latest/querying/functions/#rate. Accessed: 2019-04-28.

[19] The Linux Foundation. Querying Prometheus. https://prometheus.io/docs/prometheus/latest/querying/basics/. Accessed: 2019-04-28.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 5, pages 326–337. Addison-Wesley, 1 edition, 1994.

[21] Google. ListenableFuture. https://github.com/google/guava/wiki/ListenableFutureExplained. Accessed: 2019-04-28.

[22] Google. Micrometer Application Monitoring. Concepts. Naming Meters. https://micrometer.io/docs/concepts#_naming_meters. Accessed: 2019-04-28.

[23] Google. Protocol Buffers. https://developers.google.com/protocol-buffers/. Accessed: 2019-04-28.

[24] Red Hat. Package org.hibernate.cache. https://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/cache/package-summary.html. Accessed: 2019-04-28.

[25] Internet Engineering Task Force (IETF). The WebSocket Protocol. https://tools.ietf.org/html/rfc6455. Accessed: 2019-04-28.

[26] Graylog Inc. Graylog. https://www.graylog.org/. Accessed: 2019-04-28.

[27] Graylog Inc. Graylog Searching Documentation. https://docs.graylog.org/en/3.0/pages/queries.html. Accessed: 2019-04-28.

[28] Pivotal Software Inc. io.micrometer.core.instrument Counter. https://github.com/micrometer-metrics/micrometer/blob/master/micrometer-core/src/main/java/io/micrometer/core/instrument/Counter.java. Accessed: 2019-04-28.

[29] Pivotal Software Inc. io.micrometer.core.instrument Gauge. https://micrometer.io/docs/concepts#_gauges. Accessed: 2019-04-28.

[30] Pivotal Software Inc. io.micrometer.core.instrument Gauge. https://github.com/micrometer-metrics/micrometer/blob/master/micrometer-core/src/main/java/io/micrometer/core/instrument/Gauge.java. Accessed: 2019-04-28.

[31] Pivotal Software Inc. io.micrometer.core.instrument Timer.Sample. https://github.com/micrometer-metrics/micrometer/blob/master/micrometer-core/src/main/java/io/micrometer/core/instrument/Timer.java. Accessed: 2019-04-28.

[32] Pivotal Software Inc. Micrometer. https://micrometer.io/. Accessed: 2019-04-28.

[33] Pivotal Software Inc. Micrometer Metrics: Timers. https://github.com/micrometer-metrics/micrometer-docs/blob/master/src/docs/concepts/timers.adoc. Accessed: 2019-04-28.

[34] Pivotal Software Inc. Project Reactor. https://projectreactor.io/. Accessed: 2019-04-28.

[35] Pivotal Software Inc. Spring AOP. https://docs.spring.io/spring/docs/2.5.x/reference/aop.html. Accessed: 2019-04-28.

[36] Pivotal Software Inc. Spring Boot. https://spring.io/projects/spring-boot. Accessed: 2019-04-28.

[37] Pivotal Software Inc. Spring Boot Actuator. https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-actuator. Accessed: 2019-04-28.

[38] Pivotal Software Inc. Spring Boot Admin. https://github.com/codecentric/spring-boot-admin. Accessed: 2019-04-28.

[39] Pivotal Software Inc. Spring framework, package org.springframework.cache. https://github.com/spring-projects/spring-framework/tree/master/spring-context/src/main/java/org/springframework/cache. Accessed: 2019-04-28.

[40] Web Performance Inc. Load testing case studies. https://www.webperformance.com/library/casestudies/. Accessed: 2019-04-28.

[41] Web Performance Inc. Load testing with 5,000,000 concurrent users. https://www.webperformance.com/load-testing-tools/blog/2016/07/load-testing-with-millions-of-concurrent-users/. Accessed: 2019-04-28.

[42] Web Performance Inc. Sharepoint™ load testing services case study. https://www.webperformance.com/library/casestudies/sharepoint/index.html. Accessed: 2019-04-28.

[43] Web Performance Inc. Site is slow under load, but the servers aren't busy. https://www.webperformance.com/library/casestudies/sharepoint/index.html. Accessed: 2019-04-28.

[44] CMMI Institute. CMMI. https://cmmiinstitute.com/cmmi. Accessed: 2019-04-28.

[45] Kohsuke Kawaguchi. Jenkins. https://jenkins.io/. Accessed: 2019-04-28.

[46] Ed Korthof. Asana blog: Details on the January 22 outage. https://gatling.io/. Accessed: 2019-04-28.

[47] Ed Korthof. Facebook: More Details on Today's Outage. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919/. Accessed: 2019-04-28.

[48] Ed Korthof. Facebook on Twitter about 2019 outage. https://twitter.com/facebook/status/1106229690069442560. Accessed: 2019-04-28.

[49] James F. Kurose and Keith W. Ross. *Computer Networking*, chapter 48, page 50. Pearson, 6 edition, 2011.

[50] Grafana Labs. Grafana. https://grafana.com/. Accessed: 2019-04-28.

[51] Redis Labs. Redis. https://redis.io/. Accessed: 2019-04-28.

[52] Zabbix LLC. Zabbix. https://www.zabbix.com/. Accessed: 2019-04-28.

[53] Testing Performance Ltd. Spike Testing And Load Testing Services. http://www.testingperformance.org/definitions/what-is-spike-testing. Accessed: 2019-04-28.

[54] J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications*. Microsoft Press, 2007.

[55] J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications*, chapter 7, page 89. Microsoft Press, 2007.

[56] J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications*, chapter 16, pages 185–204. Microsoft Press, 2007.

[57] Ian Molyneaux. *The Art of Application Performance Testing*. O'Reilly Media, 2 edition, 2015.

[58] Ian Molyneaux. *The Art of Application Performance Testing*, chapter 3, pages 50–51. O'Reilly Media, 2 edition, 2015.

[59] Ian Molyneaux. *The Art of Application Performance Testing*, chapter 4, page 66. O'Reilly Media, 2 edition, 2015.

[60] Ian Molyneaux. *The Art of Application Performance Testing*, chapter 3, pages 37–43. O'Reilly Media, 2 edition, 2015.

[61] Ian Molyneaux. *The Art of Application Performance Testing*, chapter 3, pages 31–35. O'Reilly Media, 2 edition, 2015.

[62] Ian Molyneaux. *The Art of Application Performance Testing*, chapter 3, page 25. O'Reilly Media, 2 edition, 2015.

[63] Ian Molyneaux. *The Art of Application Performance Testing*, chapter 4, pages 72–73. O'Reilly Media, 2 edition, 2015.

[64] Ian Molyneaux. *The Art of Application Performance Testing*, chapter 5, pages 91–116. O'Reilly Media, 2 edition, 2015.

[65] Oracle. Dynamic Performance (V$) Views. https://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_2113.htm#REFRN30246. Accessed: 2019-04-28.

[66] Oracle. Interface MeterRegistry. https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/InvocationHandler.html. Accessed: 2019-04-28.

[67] Oracle. Monitoring Real-Time Database Performance. https://docs.oracle.com/cd/B28359_01/server.111/b28275/tdppt_realtime.htm#TDPPT033. Accessed: 2019-04-28.

[68] Oracle. Object.notify(). https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notify--. Accessed: 2019-04-28.

[69] Oracle. Object.wait(). https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait--. Accessed: 2019-04-28.

[70] Oracle. Oracle Database, V$SQL. https://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_2113.htm#REFRN30246. Accessed: 2019-04-28.

[71] Oracle. Oracle Enterprise Manager. https://www.oracle.com/enterprise-manager/. Accessed: 2019-04-28.

[72] Oracle. Oracle Java SE Licensing FAQ. https://www.oracle.com/technetwork/java/javase/overview/oracle-jdk-faqs.html. Accessed: 2019-04-28.

[73] Oracle. Package java.util.concurrent.atomic. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html. Accessed: 2019-04-28.

[74] Oracle. VisualVM. https://visualvm.github.io/. Accessed: 2019-04-28.

[75] Kaarel Purde. Koormustestimise protsessi väljatöötamine ja läbiviimine Maakleri rakenduse näitel, 2016. Bachelor's thesis.

[76] Viktor Reinok. A framework for empirical evaluation of java application performance. Master's thesis, Tallinn University of Technology, 2016.

[77] The JUnit Team. JUnit 5. https://junit.org/junit5/. Accessed: 2019-04-28.

[78] TIOBE. TIOBE index of Java 2001 - 2019. https://www.tiobe.com/tiobe-index/java/. Accessed: 2019-04-28.

[79] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 2010.

[80] Maciej Zaleski. JMeter - WebSocket Sampler. https://github.com/maciejzaleski/JMeter-WebSocketSampler. Accessed: 2019-04-28.

[81] ZeroTurnaround. XRebel. https://zeroturnaround.com/software/xrebel/. Accessed: 2019-04-28.