Masaryk University
Faculty of Informatics

Performance testing of Messaging Protocols

Bachelor's Thesis

Dominik Stuchlík

Brno, Fall 2019

Masaryk University
Faculty of Informatics

Performance testing of Messaging Protocols

Bachelor's Thesis

Dominik Stuchlík

Brno, Fall 2019

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Dominik Stuchlík

Advisor: Ing. Zdeněk Kraus

Acknowledgements

I'd like to thank my supervisor, Ing. Zdeněk Kraus, for the support and countless helpful advice and suggestions for my bachelor thesis. I also like to thank Bsc. Otavio Rodolfo Piske, my colleague from Red Hat, for his advice and support throughout the phase of implementation. Last but not least I would like to thank whole Messaging QE team from Red Hat for their help and patience.

Abstract

One of the biggest priorities of modern software development is performance testing of the finished product. The common goal of the performance testing is to ensure that the application runs fast enough to keep the attention of users, even with the unexpected amount of clients using the application at the same time. Without this type of testing, we can miss significant bugs that can be very frustrating for an end user and can lead to possible loss of customers and also huge money investment in case of fixing the bugs.

This thesis focuses on performance testing of messaging clients with protocol AMQP (Advanced Message Queuing Protocol) and a minimum of two different API (Application Programming Interface) against the broker.

The performance testing tools that are used in this thesis are called Maestro and Quiver. Maestro does not support clients testing. Quiver, on the other hand, fully supports it. Because of that Maestro has Quiver Agent, which is used to run Quiver within Maestro environment so we can test clients using Maestro. What is missing is a converter for results from Quiver. Maestro does not understand the representation of quiver output.

First chapter of this thesis describes the fundamentals of testing with a focus on performance testing. The rest of the thesis is about performance tools mentioned above, the implementation of the converter and its significant help with better automation of performance testing of messaging clients and a testing of the clients with the newly implemented converter. The results from the testing are successfully collected and appropriately evaluated with possible suggestions for improvements, which can help refine the performance of the messaging clients.

Keywords

testing, performance testing, AMQP, network technologies, Maestro, Quiver, converter

Contents

Introduction **1**

1 Software Testing Fundamentals **3**
1.1 *The Main Idea Behind Testing* 3
1.2 *Verification and Validation* 6
1.3 *Metrics* 7
1.4 *Test Types* 7
1.5 *Testing Types, Techniques and Tactics* 12

2 Performance Testing **15**
2.1 *Performance Testing Process* 15
2.2 *Performance Testing Types* 18
2.3 *Performance Testing Metrics* 19

3 Performance Tools **23**
3.1 *Maestro* 23
3.2 *Quiver* 26

4 Quiver Results Converter Implementation **29**
4.1 *Problem Introduction* 29
4.2 *Maestro Data Files* 32
4.3 *Solution* 35

5 Testing **37**
5.1 *Environment* 37
5.2 *Tests* 38
5.3 *Data Collection and Evaluation* 39

6 Conclusion and Future Work **47**

Bibliography **49**

List of Tables

1.1 Price for fixing a bug during different phases [4] *4*

3.1 Environment variables for Maestro Quiver Agent configuration [18] *26*

List of Figures

1.1 Patton's graph [6] *4*
1.2 Optimal level of testing [6] *5*
1.3 The difference between Validation and Verification *6*
1.4 V-model representing a development process [12] *11*
2.1 Performance test process [16] *17*
3.1 Maestro architecture. The arrows represent communications between the components and the SUT. The line value represents the number of connections where default is 1 [1]. *24*
4.1 Web UI cannot visualize latency or Service time statistics *30*
4.2 Web UI cannot visualize the sender and the receiver statistics *31*
4.3 Archived report files from Quiver *32*
5.1 Distribution of the Latency Percentiles *39*
5.2 Service Time Statistics *39*
5.3 Sender Rate Distribution Over Time *40*
5.4 Receiver Rate Distribution Over Time *40*
5.5 Sender Rate Statistics *41*
5.6 Receiver Rate Statistics *41*
5.7 Distribution of the Latency Percentiles *42*
5.8 Service Time Statistics *42*
5.9 Sender Rate Distribution Over Time *43*
5.10 Receiver Rate Distribution Over Time *43*
5.11 Sender Rate Statistics *44*
5.12 Receiver Rate Statistics *44*

Introduction

One of the biggest priorities of modern software development is performance testing of the finished product. The primary goal of this type of testing is to test the system as a whole, how it behaves in boundary conditions, how it endures when many users are using it. It also checks if the system handles various situations appropriately and if the system meets customer requirements. If the finished product is frustrating for the customer, it can mean losing customer's trust in the brand and losing money on fixing the issues, which, in some cases, can have a massive impact on the whole company. There is a significant stress on performance testing, for example, in medicine, where every single device or a tool must be 100% functional in every condition. If not, the consequences can be fatal. [1]

Nowadays, developers can find many alternatives on how to correctly and quickly test the product developed. The developer is offered different tools, for example, SmartMeter.io [1], Apache JMeter [2], SpecJMS [3] and many others. The main performance testing tools that Red Hat develops and uses for messaging products are Maestro and Quiver. Maestro is a distributed performance testing tool that is opinionated towards large scale performance test execution and automation [2]. Quiver is a set of tools for testing the performance of messaging clients and servers [3]. *Quality Assurance* (QA) engineers from messaging department use both of these tools for performance tests automation. These tools are managing complex back-end and allow the developer to focus on test functionality and fulfillment of system requirements. This level of management saves the developer a lot of time and also a lot of company money for the testing.

---

[1] `https://www.smartmeter.io/`
[2] `https://jmeter.apache.org/`
[3] `https://www.spec.org/jms2007/`

This thesis focuses on performance testing of messaging clients for network protocol *Advanced Message Queuing Protocol* (AMQP), developed by Red Hat Inc. The first half of the thesis focuses on the technical side, such as fundamentals of software testing, performance testing, performance tools, and protocol AMQP. The other half focuses on the implementation of the result converter for Maestro, on testing itself, the metrics and approach for further in-depth analysis. It also contains the conclusions drawn from the testing and possible improvements.

1 Software Testing Fundamentals

1.1 The Main Idea Behind Testing

A software testing as an independent department was not a reality until the early 2000s. Software developers, and not just them, realized, that testing is a crucial part of software development. There can be mistrust between the company and customers because of a lot of bugs and issues within the finished product. The answer for this is a team of testers, which can expose the defects and fix those before it gets to the customer. The team is called Quality Assurance team, or QA team for short, and its primary goal is to assure quality in the development process. The team is not only responsible for testing but also compliance of standards and procedures used in development. The QA team's job is also to improve corporate regulations and tools used for testing. This chapter is based on knowledge available in [4].

The Cost of Finding a Bug

To err is human, and we are not always able to avoid them altogether. For that, we have a testing process, in which we can detect bugs before they get to the customer. Testing itself can help save a lot of company's money and also deepen customer's trust in its products. However, it is not inexpensive.

Patton graph 1.1 of bug fix costs well shows why in most organizations, the costs of testing can be from 30-50% of the budget. This graph is reflecting *Boehm's first law*:

> Errors are most frequent during the requirements and design activities and are the more expensive, the later they are removed. [5]

The table 1.1 shows us in detail the cost ratio of fixing a bug throughout all phases. Note to be given is that with increasing options for software distribution, such as automatic updates, the cost for post-release is decreasing. [7]

Let us have a small example – if the bug appeared in the Conception phase and it was detected during the Testing phase, the fix will be

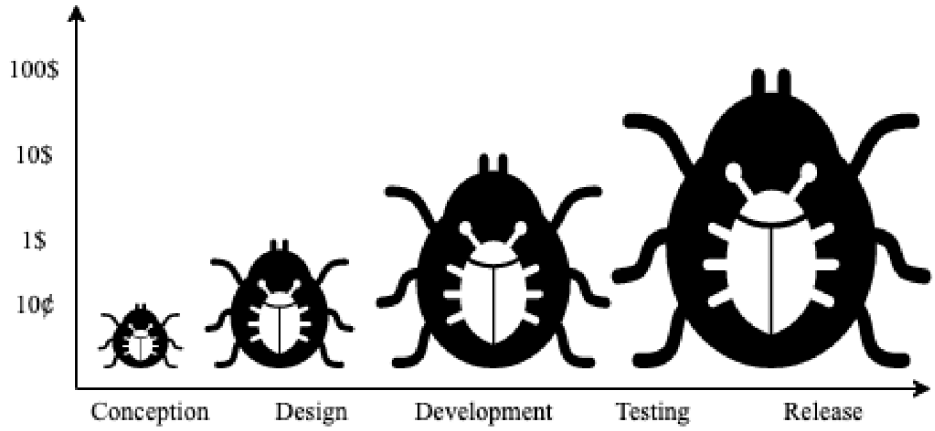


Figure 1.1: Patton's graph [6]

Table 1.1: Price for fixing a bug during different phases [4]

Price for fixing a bug (numbers meaning multiplications)		The phase when the bug was detected				
		Conception	Design	Development	Testing	Release
The phase when the bug appeared	Conception	1	3	5–10	10	10–100
	Design	–	1	10	15	25–100
	Development	–	–	1	10	10–25

10× more expensive than if it was detected during the Conception phase.

As we can see, the cheapest way to get rid of bugs is during the Conception or Design phase. However, fixing bugs in the Development phase is relatively cheap as well. For that, the developer tests are one of the most important as they can save a lot of money.

When to Stop with Testing

Software testing and development has also an economic side of view. *Quality, Scope, Time, Budget* (QSTB) are parameters, where each parameter needs to be consistent with the other three. Besides that it has to comply with the business plan of the project.

Developers should stop with testing when the price for fixing a bug is the same or bigger then a price for leaving a bug there.

On the contrary there are scenarios when developers cannot afford to leave any bugs in the product. For example in pharmaceutical

industry where every single tool or machine has to be 100% reliable or the consequences can be fatal.

Patton is introducing in his graph 1.2 the relationship between expenditure and number of tests. Every software project has its own optimal level of testing.

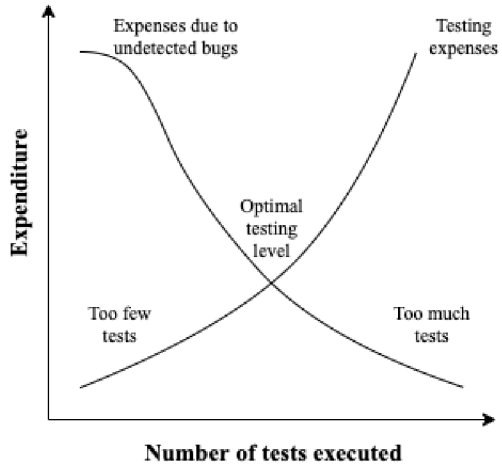


Figure 1.2: Optimal level of testing [6]

Considering the number of different test types and different test sets that testers can come up with there is more specific description of the graph – every software project has optimal level of testing **for every test set**.

The Ability to Change

Next to the cost of finding a bug the ability to change is another very substantial aspect. The ability to change is decreasing as the software evolves or the developers are changing. The best way to avoid the project going to stagnation are automated tests. The stagnation of a project means that the project got to a state that even minor fixes can break something else. These dangerous bugs are technically called *regression bugs*. Testing the full product at this point is very complicated. If the developer has well written automated tests with good coverage, there is no need to fear the stagnation consequences.

Without a wide variety of automated tests, the project from certain size is destined to stagnate, because any change is risky and expensive.

Because of this for many project managers is more important the ability to change then the cost of finding a bug.

Most importantly testing cannot tell one if the program is without any bugs.

> Program testing can be used to show the presence of bugs, but never to show their absence! [8]

1.2 Verification and Validation

Both verification and validation are an essential part of testing. In picture 1.3, there is the main difference between these two approaches.

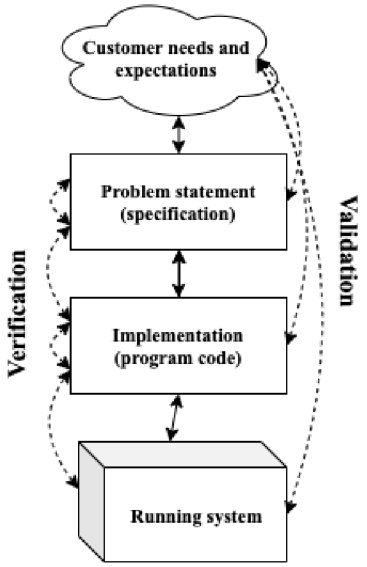


Figure 1.3: The difference between Validation and Verification

According to *Institute of Electrical and Electronics Engineers* (IEEE) Standard Glossary of Software Engineering Terminology [9], **Verification** is a process of evaluating a system or component to determine whether the system of a given development phase satisfies the conditions imposed at the start of that phase. The fundamental question is: "Are we building the system right (according to specification)?"

About **Validation** the IEEE [9] says that it is a process of evaluating a system or component during or at the end of the development pro-

cess to determine whether a system or component satisfies specified requirements. The fundamental question is: "Are we building the right system?"

Validation and verification complement each other hence the best practice is to use both of these techniques. Verification is searching for bugs according to specification but cannot assure correctness. Validation can assure correctness. The product that is functional but does not respect customer's specifications is as much useless as a product that is according to customer's specifications but has many bugs in it.

1.3 Metrics

When evaluating the quality and performance of a product, developers need results to examine. These results (metrics and criteria) quantify the level of quality. There are many metrics:

- **metrics based on bugs** [10]
  The number of bugs found is not as important as sorting the bugs according to the severity [1] and belonging to components.
- **metrics based on tests**
  Again the number of tests is not as important as their results. This type of metrics is the most important for monitoring the state of the product tested as they are based on functional tests of the product.
- **metrics based on code**
  These metrics are dependent on code coverage, which can be determined easily and with high accuracy. However, it does not have to be accurate in the functional side of the product.

1.4 Test Types

Positive and Negative Tests

Generally the tests can be divided into two groups based on the character of input data.

[1] The severity is a bug criterion that gives us the information, how much the bug influence correct use of the application or how much is the bug irritating for a user.

First of the two are **positive tests**, also known as **test-to-pass**. These tests are verifying correct behavior in the expected conditions of the application tested. The input for the tests are correct or expected values [4]:

- engineer is in a role of ordinary user,
- engineer verifies minimal requirements,
- engineer doesn't try to break the software (he doesn't go behind the boarders of software's capabilities)
- engineer applies the most basic and the most probable inputs,
- the goal is to verify expected correct functionality during usual usage.

This type of approach is in the majority of the tester community more accepted as it is more intuitive than the negative tests. The output is expected behavior or its confirmation.

The second type of approach is **negative tests**, also known as **test-to-fail**. This type of tests is focusing on verifying system behavior during abnormal or unexpected conditions. The input for the tests is incorrect or forbidden values. Engineers use these tests mainly to simulate scenarios that the user can get into. The tests are checking alternative flows in code.

Another domain of negative tests is verifying the state of the environment. For example, there are multiple reasons why the program cannot open the file that it wants and it receives unexpected or no response.

Negative tests are verifying if the program is responding correctly to these situations. Correct respond can be to print an error message or to throw an exception but mainly to continue without crashing. On the other hand, incorrect respond can be previously mentioned crash of the program, data loss and so on.

At the beginning of the development process developers mainly use positive tests as in this phase the system is crashing even with correct input. The negative tests are used after the system is more stable.

Testing Approach

The tests can be divided into a spectrum of different categories. One possible and widely used division could be to these four categories:

- static vs. dynamic testing – static testing involves verification, whereas dynamic testing also involves validation [11],
- the box approach – divided to white-box and black-box testing. The central aspect is the level of code visibility that engineer has when designing tests,
- testing levels – tests are based on the development stage of the application,
- dimensions of software quality – includes more attributes beyond functionality.

Static vs. Dynamic Testing

Static tests do not require a running program. Not only for that is possible to start with static testing during early phases of development. Static tests can be, for example, proofreading, source code inspection, or static program analysis.

Dynamic tests are always running on the executed program or its parts. [4]

The Box Approach

The level of code visibility is divided into two (sometimes three) groups.

Black box testing includes tests that are written without knowing the implementation. The specifications of the customer inspire the input of tests like this. Black box testing should simulate a user perspective. That means that black box testing is used to approach a realistic way of program use. This type of testing is typical in later stages of development.

White box, also known as *clear* or *transparent box*, testing includes tests that are written with knowing every detail of the implementation. White box testing simulates developer perspective. Engineer has full access to the source code, and for that, the testing itself is more consistent. With that said, the white box testing is often vastly different than

realistic scenarios of program use. Developers typically use this type of testing in the early stages of development (unit tests).

Gray box testing is sometimes featured as the third type of box approach. The gray box means that the engineer knows the implementation partially and with that knowledge the engineer designs the tests. However, the execution of the tests is done from a user perspective as it is in black box testing. Developers mainly use gray box testing for exception handling, performance testing, and so on.

Testing Levels

On picture 1.4, there is shown that every development level has an equivalent in testing level. Tests can be divided by levels: [4]

- unit tests
  These tests are testing the smallest parts of system (methods and their parts). In other words, the parts are tested separately.
- integration tests
  The tests are verifying correct behavior of subsystems (modules and classes). In this case, the parts are tested together.
- system tests or *System integration tests* (SIT)
  These tests contain comprehensive scenarios which are going through the whole system.
- acceptance tests or *User acceptance tests* (UAT)
  These tests are performed by a customer as part of the hand-off process.

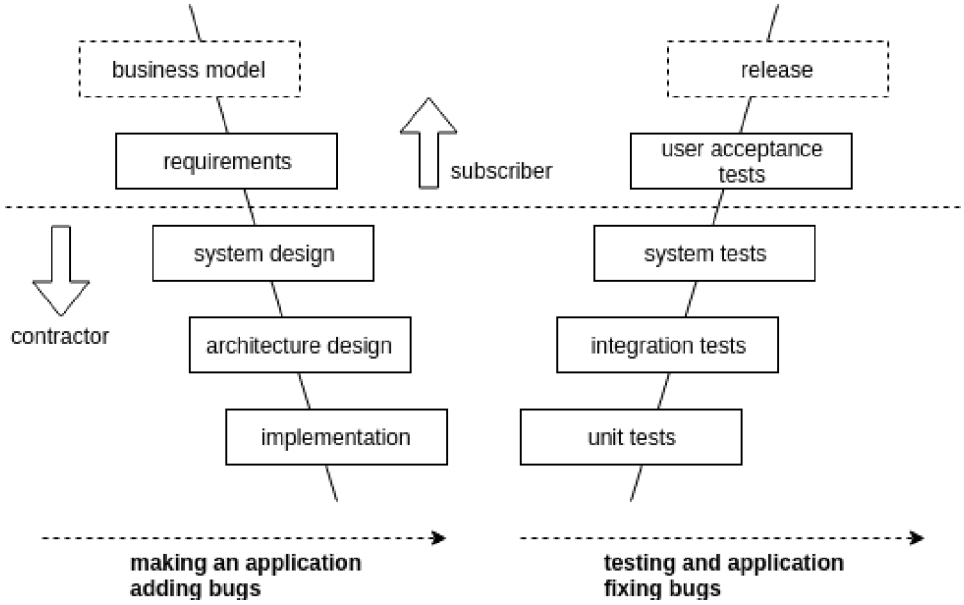


Figure 1.4: V-model representing a development process [12]

Tests can also be divided by the testing environment: [4]

- development – testing performed by developers
  - *unit tests*
  - *integration tests*
  - *Factory Acceptance Tests (FAT)*
- testing – testing performed by a team of testers
  - *smoke tests*
  - *system tests*
  - *regression tests*
- integration and system testing environment – user representatives are also involved in testing
  - *User Acceptance Tests (UAT)*
- production – testing performed by end users

– *beta tests*

The goal is to organize testing in a better manner, so the developer gets feedback as soon as possible. Excellent organization and quick feedback can save money for fixing a bug and speed up the development.

Dimensions of Software Quality

The only aspect of testing was functionality. However, it is only one of many dimensions that can be tested. The other dimensions being non-functional requirements. For quality assurance, every dimension should have test coverage consisting of at least one test.

Along with dimensions is also being mentioned an abbreviation FURPS+, which stands for: [4]

- **F**unctionality – assures correct behavior of the system,
- **U**sability – tests ease with which one can use the system,
- **R**eliability – tests if the system is behaving the same in all situations,
- **P**erformance – assures if the system is fast enough for the customer,
- **S**upportability – tests if the system is not having problems with targeted hardware or software configurations, it also tests quality of installation and other aspects related to system maintenance,
- **+** (Others) – compatibility, security, consistency, etc.

1.5 Testing Types, Techniques and Tactics

Testing with Scenarios

The testing scenarios are made based on specifications or how the user uses the application. These scenarios are consisting of step by step description and behavior expected from the application. The engineer will use the step by step guide and will look for correct behavior. This testing is also called controlled manual testing, and typically, it exposes many bugs and defects.

Exploratory Testing

This type of testing is complementary to the testing with scenarios. Just like the testing with scenarios, this type of testing tests functionality and usability of the application. What is different, however, is that the step by step guide is not established.

Regression Testing

Regression testing assures that after making changes to the code, the original functionality still behaves as it should. With this type of testing, testers can find defects that are in the application as a consequence of the changes. Typically the regression tests are any functional tests that run successfully before made changes. New or changed functionality is tested with scenarios or exploratory testing, and unchanged functionality is tested with regression tests.

Unit Testing

This testing comes right after the developer finishes with the part of the implementation. The developer finds the bug, detects the cause, and fixes the bug. It is the first stage of testing and also the cheapest one because there is no need to report the bugs anywhere, everything is done by the developer, that understands the code the best.

Beta Testing

Beta testing, or *external acceptance testing*, is usually done by external users. If a narrow circle of verified users does the testing, it is called closed beta testing. Open beta testing is open to all users. The advantage of open beta testing is a broader range of users, but on the other hand, the disadvantage of that is poorly or incomplete reports of bugs.

Smoke Testing

Smoke tests, also called *sanity tests* or *build verification tests*, are performed after the new project build. It tests if all components of the project were built correctly. If there ought to be any defects, the new

version of the project should be returned for fixing and does not have to be unnecessarily tested.

Acceptance Testing

This type of tests is also known as *UAT*. It is based on testing with scenarios that the company and the customer agreed on.

Conformance Testing

After defect removal, the part of the code that was changed should be retested to assure that the changes were valid. Therefore, this testing is also known as *retesting*. According to [4], 15-30% of defects are not fixed properly.

2 Performance Testing

Performance testing is a complex conceptual and technical activity for answering questions related to the performance of a system, also known as *System Under Test* (SUT) [1] [14].

Generally, performance testing means that engineers test the SUT under realistic conditions based on performance requirements. Performance testing comes at the end of the test plan and after functional testing. However, the performance evaluation is crucial in early stages of the development, when main architectural choices are made [15].

With the use of performance tests, engineers can find out what is the responsiveness of the system or how fast it can accomplish tasks under various loads. The tests also can uncover possible degradation of performance with time, software crashes, or errors related to loading time.

According to [16], performance testing can fit into three fundamental objectives. The first objective is to assure that the system satisfies performance requirements. The second is to find out the boundary limits and capacity of the system. This information helps the customer in searching for the best solution for their budget and requirements. The third and last objective is to help developers with finding performance issues and further improve the performance of the system.

2.1 Performance Testing Process

A major focus in performance testing is to assure the following system abilities [17] [1]:

- **Processing speed and response time** – system should react fast under any load acceptable and ideally with the lowest possible response time.
- **Throughput** – system should bear as many transactions, messages or events as required.

[1] System Under Test is a type of test object that is a system. [13] In the other words, it is the system being tested.

- **Reliability** – system should be able to perform all of its tasks under system load for an acceptable amount of time.
- **Scalability** – system should be able to handle increasing amounts of workload (such as network traffic load, input or output throughput, and data and transaction volume) by adding more resources to the system.
- **Availability** – system should be available to all its users with all its functions under acceptable system load.
- **Utilization** – system should utilize given resources efficiently according to the requirements.

Figure 2.1 represents performance testing process consisting of working steps similar to the development process. The performance test process starts with understanding the performance requirements, works down to the design, implementation and execution of the tests, and ends with results evaluation.

The first step for an engineer is to select performance requirements that describe the best the SUT. The engineer has to fully understand the requirements of the customer, which can be sometimes unclear. Hence, it is suitable for an engineer to refer to system specifications or to discuss the performance requirements with project manager or customer directly.

The outcome of this step is the specification of necessary performance requirements that are measurable and achievable.

The steps two through six are focused on test planning. The planning includes declaring major focus points of performance testing, define performance evaluation metrics as well as performance test metrics and required solutions. In the test planning phase engineer should also choose suitable scenarios and select performance tools necessary for monitoring, data collection, and analysis.

The result of this step is a performance test plan with a description of all the tasks mentioned above with hardware specification, timelines, and performance test schedule also included in the document.

The seventh step focuses on the design and implementation of test cases or test suites as well as performance test tool configuration or development.

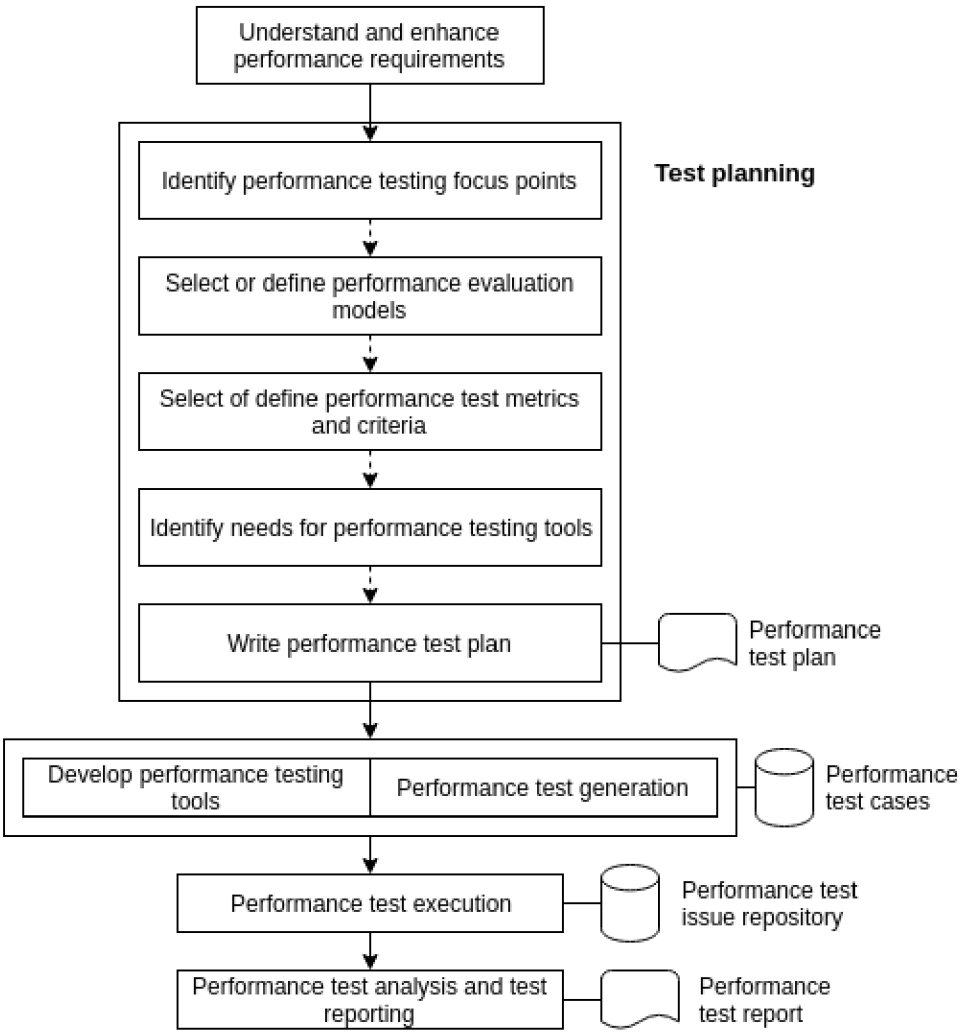


Figure 2.1: Performance test process [16]

The eighth step consists of test execution and results evaluation. The outcome of this step is information needed for in-depth analysis and documentation in the last step of the testing process.

The ninth step is a final analysis and report of the tested system. Result of this step is detailed report including evaluation and validation of the system gathered from performance tests' results. The report precisely specifies issues, boundary limits, and capabilities of the sys-

tem. Moreover, various graphs are included displaying performance metrics along the duration of test execution.

2.2 Performance Testing Types

There are several variations of performance test types. The correct test type is designated by the testing requirements, system dispositions, or time given for the performance testing. Here is a list of basic and well-known testing types.

Load Testing

Load testing is a testing method that simulates diverse workloads and observes how the system behaves. This type of testing can detect problems or issues of the SUT such as insufficient performance when the number of users increases, the availability and scalability issues, or abnormal delays. A load test simulates real-life workloads to test system stability, limits, or thresholds.

Stress Testing

Stress testing is a particular type of load testing that focuses more on unexpected workload measurement. Stress tests get the system under stressful conditions such as short bursts of a massive amount of load, load tests with an enormous number of users or continuous sending of requests. These stressful conditions allow engineers to find system limits and make sure that the system can recover after the stress.

Endurance Testing

Endurance or *soak* testing is used to test the stability and reliability of the system over extended period of time. The typical issue that the soak tests can uncover are memory leaks, which in short runs may not be visible but over the time it can lead up to massive memory usage or leaks and possible crash of the SUT.

Scalability Testing

Scalability testing focuses on system behavior under increasing load when the resources vary. The results determine how much the system performance improves when comparing, for example, if the system runs on a machine with one CPU against running on a machine with two CPUs.

Volume Testing

Volume testing is a testing method that considers a volume of a load and detects if the SUT appropriately handles this amount of load. For example, a client-server application is tested for the maximum number of users it can handle. The difference between this type of testing and load testing lies in focus on the load amount because volume testing does not imply that the load is very high.

Benchmarking

Benchmarking is used for assuring smooth running of the system on different hardware. It is considered in general to be a load test which has a well-defined execution procedure that ensures reproducible results. These results can define minimal hardware requirements and also verify optimal hardware configuration.

2.3 Performance Testing Metrics

As is the case of software testing, there are a lot of metrics engineers can track. Each metric can have different priority based on system requirements. These metrics determine the data that are collected, analyzed, and reported pursuant the system performance or one of its components. Here is a list of standard metrics used during performance testing.

Throughput

Throughput is a metric, which refers to the number of requests per second that the system can handle [1]. From a network perspective, the

throughput can represent the rate of successfully delivered messages over the communication channel. According to [1], throughput is measurable by load testing with continuously raising the load to find out when the response takes longer than the acceptable threshold.

Response Time and Latency

One of the most common performance metrics is response time. It portrays the time that SUT needs to accept the user's request and respond to that request. Users may find the system very irritating if the response time is too long. Therefore, testing this metric is very important as it can induce the user to look elsewhere. Response time consists of two parts – service time and latency.

Service Time Service time represents the time that takes the system to evaluate a request from the user and send the response back to the user. The period starts after the system received the request and ends right after the system sending the response back.

Latency Latency describes the delay involved in the time from sending the user's request to receiving it by the server. Latency can increase based on the location of the client and the server, which can be, for example, on different sides of the planet.

Error Rate The Error Rate is the mathematical calculation that produces a percentage of problem requests compared to all requests. [1] Problem requests can be found, for example, in communication between client and server where some requests can get lost in the network or get corrupted by another device or signal. This problem can result in lower performance as the damaged requests must be resent.

Resource Usage Resource usage describes the utilization of the resources that are allocated for the task or the system. These resources consist of:

- CPU utilization
- Memory utilization
- Network utilization
- Operating System limitations
- Disk usage

Nowadays, great emphasis is placed on this metric as it can lead to the total system failure if the system has not enough resources or using them ineffectively.

3 Performance Tools

3.1 Maestro

Maestro Architecture and Overview

Maestro is a distributed performance testing tool that is opinionated towards large scale performance test execution and automation [2].

Picture 3.1 shows, how architecture of Maestro tool looks like. Maestro consists of clients as a front-end, Maestro Broker as a message director and sender, receiver, inspectors and agents as back-end.

Maestro focuses on multi-node deployment. It coordinates these nodes to generate load and use this to simulate the workload for the SUT. Running large scale performance tests can be tedious so preferably one can use an container-orchestration system such as Kubernetes [1], Origin Community Distribution of Kubernetes [2] or Red Hat OpenShift [3].

However, for small scale or local tests, there is no need to use orchestration systems. Maestro supports docker containers that should be sufficient. It comes with performance penalty cost of 10% of the max throughput.

The nodes, also known as backends, use the *Protocol Under Test* (PUT) for generating workload for SUT. The PUT can be any of the supported protocols.

---

[1] Kubernetes - `https://kubernetes.io/`
[2] OKD - `https://www.okd.io/`
[3] OpenShift - `https://www.openshift.com/`

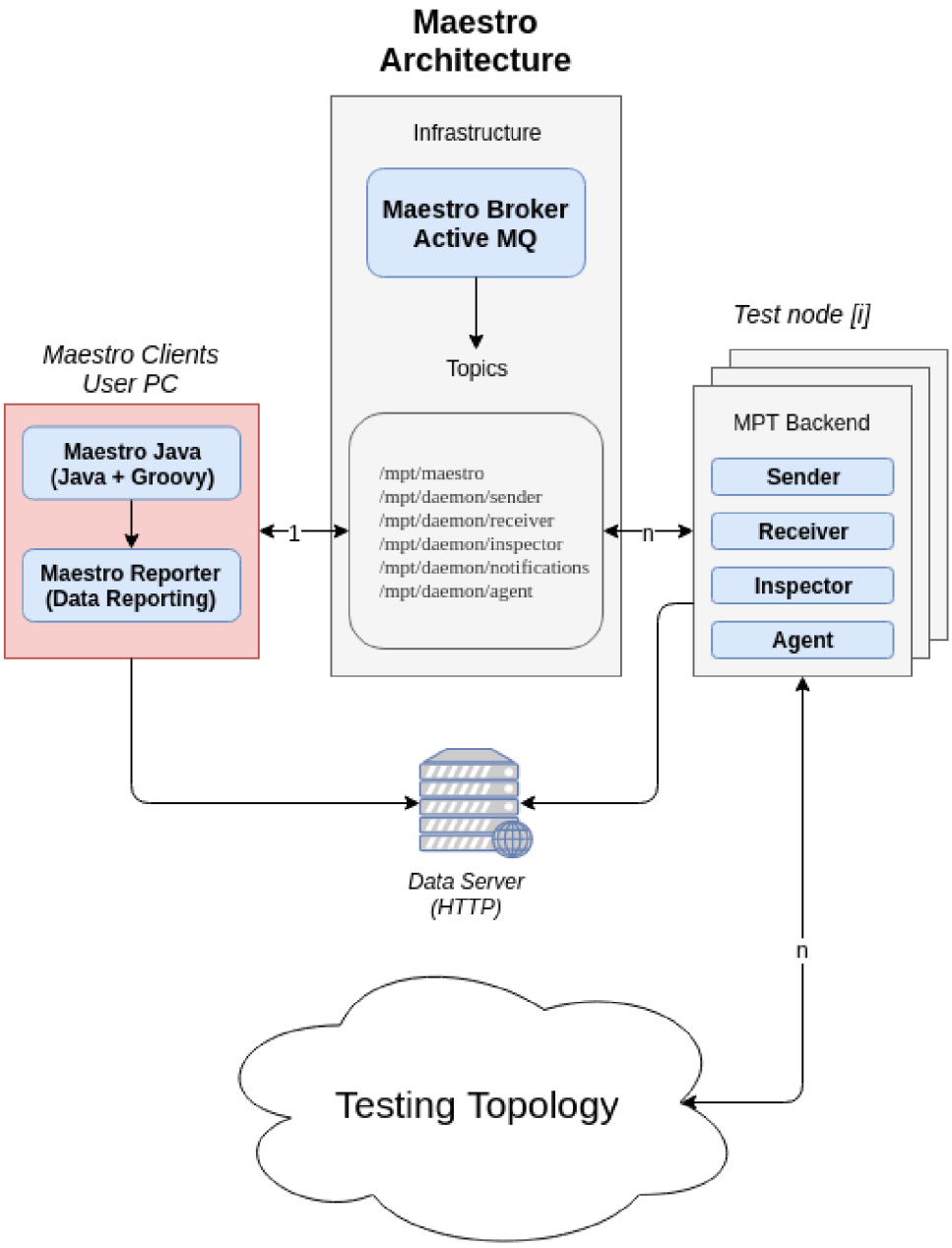


Figure 3.1: Maestro architecture. The arrows represent communications between the components and the SUT. The line value represents the number of connections where default is 1 [1].

Docker Compose deployment and usage

Docker Compose [5] is required for this way of deployment. In this thesis Docker Compose is used to deploy Maestro. Main reason being that it is very easy to deploy with Docker and with only a small price of 10% off the max throughput.

A user can use `docker-compose` command to specify how many workers, agents, inspectors and other back-ends should the deployment have or even use some of the deployment templates included in Maestro.

Docker Compose then creates a network also called a cluster. After the cluster is up user can launch number of containers on which the user can run Maestro tests.

The approach used in this thesis was very simple cluster with only one agent. Then using `docker run` command the maestro-client is initiated from which the user can run the tests themselves.

The reports can be visible on `http://localhost:6500`.

There is also another more preferable way to deploy Maestro tool and that is by using Kubernetes. More information about Kubernetes deployment and deployment using `docker-compose` are available in the documentation[4].

Maestro Quiver Agent

Maestro Quiver Agent [6] is used for running Quiver tests within Maestro performance testing tool. Whilst Maestro is responsible for test orchestration, Quiver runs the tests themselves.

More information about Quiver is available in the section 3.2.

There are differences between Maestro and Quiver implementations. Main difference being that Maestro is not able to parse and appropriately represent the data that Quiver returns as a result of the testing.

Table 1.1 shows the environment variables that are used to configure Maestro Quiver Agent.

---

[4] `https://github.com/maestro-performance/maestro-java/blob/master/extra/doc/Deployment.md`.

[5] Docker Compose - `https://docs.docker.com/compose/`

[6] https://github.com/maestro-performance/maestro-quiver-agent

Table 3.1: Environment variables for Maestro Quiver Agent configuration [18]

Variable Name	Default Value	Description
SEND_RECEIVE_URL	null	The URL of the SUT
MAESTRO_BROKER	null	The URL of the Maestro broker
SOURCE_URL	null	URL of the Maestro Quiver Agent GitHub
SOURCE_BRANCH	master	Git branch to use for the source URL
TEST_DURATION	null	Estimated test duration

Maestro Quiver Agent is easily deployable by using `docker-compose`:

```
docker-compose -f docker-compose up \
-d --scale agent=1
```

To run Quiver tests, user has to run *FlexibleTest* script from Maestro version 1.5 or newer.

Maestro Quiver Agent can also be used with OpenShift. More information on the agent and OpenShift are here [6].

3.2 Quiver

Overview

The Quiver is used to test the performance of messaging clients and servers. It consists of implementations, also known as *arrows*, of clients written in several programming languages. The arrows can either send or receive messages and send the transfer data to standard output.

There are two main commands. First is `quiver-arrow`, which runs a single implementation as a sender or receiver and shows its output. With this command, the user can set the arrow's implementation, configure reporting, or test execution parameters. The second command

[6] https://github.com/maestro-performance/maestro-quiver-agent

is `quiver`, which is used as a launcher for the sender and receiver and provides the summary of the transmission.

There is also a `quiver-server` command to start a server on the given address.

Deployment and usage

The Quiver has a straight forward installation and deployment. Users can either install the Quiver from the repository or install the Quiver from sources, which is very easy and intuitive with the use of `make` building tool.

Running this tool, the user will need to run an instance of ActiveMQ or ActiveMQ Artemis broker. Quiver can run, for example, like this:

```
quiver --sender qpid-jms \
--receiver qpid-proton-python --duration 10 q0
```

The command above defines the sender and receiver implementation, sets the duration of the test execution, and specifies the broker queue to work with.

More information on the installation and use of the Quiver tool are available here [7].

[7] https://github.com/ssorj/quiver

4 Quiver Results Converter Implementation

This chapter introduces the Quiver Results Converter – standalone CLI utility for data conversion.

4.1 Problem Introduction

Performance testing of messaging clients has always been done by Quiver Test Tool, whereas the more capable Maestro tested the rest of the ActiveMQ components such as a broker. Then the maestro-quiver-agent was created as an endpoint for Maestro with the ability to run Quiver in itself. With this client, there is no need to implement new functionality to Maestro to be able to run performance tests for messaging clients. It just runs the Quiver within the agent and takes care of the orchestration. The Quiver will do all the testing.

Due to the different implementations of both tools, the Maestro is not able to correctly display the results from Quiver as seen in 4.1 and 4.2. Maestro uses its own *Maestro Binary Rate Format*, which is described in section 4.2 while Quiver uses simple *Comma-separated values* (CSV) files. The Maestro at least provides the user with output files from Quiver as seen in 4.3.

Distribution of the Latency Percentiles

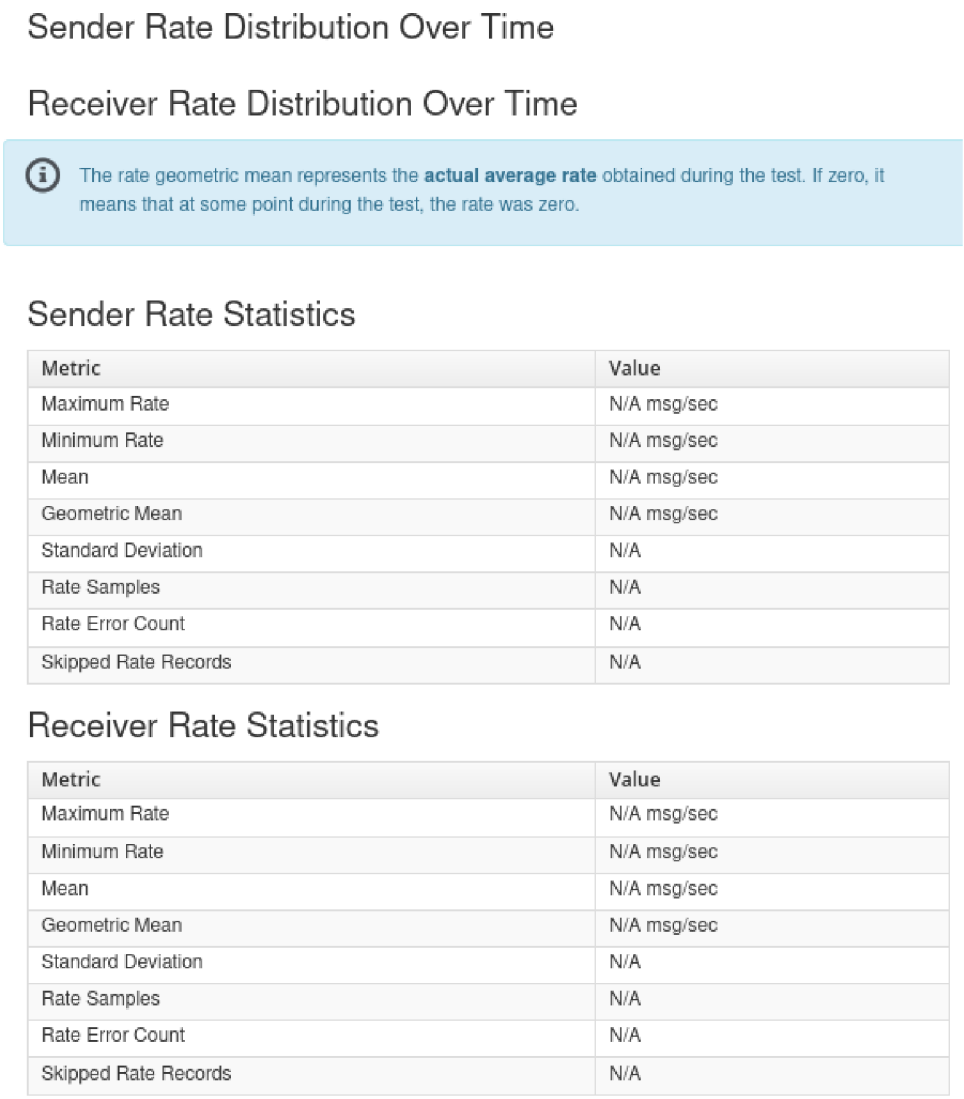
Service Time Statistics

Metric	Value
Total Count	N/A
Max Latency	N/A µs
99.99th Percentile	N/A µs
99.9th Percentile	N/A µs
99th Percentile	N/A µs
95th Percentile	N/A µs
90th Percentile	N/A µs
50th Percentile	N/A µs

Response Time Statistics

Metric	Value
Total Count	N/A
Max Latency	N/A µs
99.99th Percentile	N/A µs
99.9th Percentile	N/A µs
99th Percentile	N/A µs
95th Percentile	N/A µs
90th Percentile	N/A µs
50th Percentile	N/A µs

Figure 4.1: Web UI cannot visualize latency or Service time statistics



Sender Rate Distribution Over Time

Receiver Rate Distribution Over Time

The rate geometric mean represents the **actual average rate** obtained during the test. If zero, it means that at some point during the test, the rate was zero.

Sender Rate Statistics

Metric	Value
Maximum Rate	N/A msg/sec
Minimum Rate	N/A msg/sec
Mean	N/A msg/sec
Geometric Mean	N/A msg/sec
Standard Deviation	N/A
Rate Samples	N/A
Rate Error Count	N/A
Skipped Rate Records	N/A

Receiver Rate Statistics

Metric	Value
Maximum Rate	N/A msg/sec
Minimum Rate	N/A msg/sec
Mean	N/A msg/sec
Geometric Mean	N/A msg/sec
Standard Deviation	N/A
Rate Samples	N/A
Rate Error Count	N/A
Skipped Rate Records	N/A

Figure 4.2: Web UI cannot visualize the sender and the receiver statistics

Figure 4.3: Archived report files from Quiver

4.2 Maestro Data Files

Maestro captures all information from the testing and stores them to multiple files for the sender and receiver separately. This section is inspired by information gathered from [2].

Sender

- *sender.dat* – rate data from sender stored in Maestro Binary Rate Format
- *test.properties* – includes all test properties (duration, message size, etc.)
- *system.properties* – includes all system properties (OS architecture, number of CPUs, etc.)

Receiver

- *receiver.dat* – rate data from receiver stored in Maestro Binary Rate Format
- *test.properties* – includes all test properties (duration, message size, etc.)

- *system.properties* – includes all system properties (OS architecture, number of CPUs, etc.)

- *receiverd-latency.hdr* – data about latency distribution during the test

Maestro Binary Rate Format

The Maestro Binary Rate Format is used to store the rates recorded during the test. [2] The format consists of **Header** and **Records**.

The Header consists of metadata needed to determine the file format and its contents. The metadata is the following:

- Format Name – identifies the file format; "maestro" is default value

- Format Version – indicates the format version; "1" is current version

- Maestro Version – indicates the version of Maestro; "155" is 1.5.5 version of Maestro

- Role – number representing the role code documented in Maestro Orchestration Protocol [1]

After the header come one or more records. Each record consists of 3 fields:

- Metadata – placeholder for record metadata; currently "0" is default value

- Count – a number of messages exchanged per second

- Timestamp – a time of the data collection with microseconds precision

[1] https://github.com/maestro-performance/maestro-java/wiki/Maestro-Orchestration-Protocol

Test Properties

The `test.properties` file contains test properties set up by a client.
Sample of the file:

```
fcl=0
parallelCount=1
apiName=JMS
duration=1000000
limitDestinations=1
durationType=count
rate=0
variableSize=false
apiVersion=1.1
brokerUri=tcp\://sut\:61616/test.performance.queue
messageSize=100
protocol=OPENWIRE
```

System Properties

The `system.properties` file contains system properties of the peer running the test.
Sample of the file:

```
workerJvmVersion=25.191-b12
workerOperatingSystemName=Linux
workerOperatingSystemCpuCount=1
workerOperatingSystemVersion=3.10.0-957.el7.x86_64
workerOperatingSystemMemory=4026138624
workerJvmName=OpenJDK 64-Bit Server VM
workerOperatingSystemArch=amd64
workerJvmJavaHome=
    /usr/lib/jvm/ \
    java-1.8.0-openjdk-1.8.0.191.b12-1.el7_6.x86_64/jre
workerJvmJavaVersion=1.8.0_191
```

HDR Latency File

The `receiverd-latency.hdr` file contains receiver latency data. This data is stored in HDR Histogram log format [2].

4.3 Solution

Initiate thought was to implement the converter in the maestro-quiver-agent itself. However, the final decision was to create a standalone CLI utility and let the agent use it. The benefits of this implementation strategy are that the simplicity of the maestro-quiver-agent's code is preserved, and now everyone can use this utility to convert quiver results to Maestro supported format manually.

Quiver Results Converter

The Quiver Results Converter [3] is the solution to unify the performance results to one data format. It works as a standalone utility. The input for this converter should be two files consisting of the data CSV file from Quiver, `sender-transfers.csv.xz` or `receiver-transfers.csv.xz`, and as a second argument the JSON file with the summary about the peer, `sender-summary.json` or `receiver-summary.json`. The output is a bunch of generated files, according to Maestro Data Files found in section 4.2.

The converter is written in Java as it imports some of the Maestro packages, which are also written in Java. It consists of `MainCLI` class, which includes the interaction with the utility itself. The class is easily extendable. For now it includes only one function – `convert`.

During the initialization of the `QuiverResultsConverter` class, the arguments go through the process of determining whether the files are from a receiver or a sender. Then comes the decompression of the `.xz` file, and the files are set for further processing. In the end, the `run()` method of the "convert" action is initiated.

The majority of the functionality is in `QuiverResultsConverter` class. The wrapper for all the phases needed for output file generation is the method `run()` mentioned above. The basic program timeline

[2] https://github.com/HdrHistogram/HdrHistogram
[3] https://github.com/dstuchli/quiver-results-converter

is that first of all, the Maestro `BinaryRateWriter` is initialized as it is used to write the converted records to the output `.dat` file. Right after the writer's initialization, the convert phase starts. The CSV file is opened for reading and line after line the file is processed.

The Quiver stores data differently for sender and receiver. In the sender file, there is on each line the *message ID* and *timestamp* when it was sent with millisecond precision. The receiver file has three fields – *message ID*, *timestamp* when the message was sent, and *timestamp* when the message was received, both with millisecond precision.

This is where Maestro is different. First of all, every Maestro file has its Header described in section 4.2. Then each line consists of the `count of messages` sent or received within one second and the `timestamp` with microsecond precision.

The converter can correctly modify the data to match the Maestro format and store the results for further processing. Stored records are then written into the output `.dat` file.

The next file needed to be generated is `test.properties`. The information stored in this file is taken from the summary file given as a second argument to the utility. For the receiver, the converter also generates a particular file called `receiverd-latency.hdr`. The file includes information about latency.

Part of the utility is also a run script which usage is as follows:

```
./quiver-results-converter.sh <Action>

Actions:
    convert <arguments>
        Arguments being the files both with
        sender or both with receiver prefix.
    help
    version
```

5 Testing

5.1 Environment

The testing is run on a separate machine. Maestro is deployed using `docker-compose` as follows:

```
docker-compose -f docker-compose.yml \
-f docker-artemis-compose.yml up \
--scale worker=0 --scale agent=1 --scale inspector=0 -d
```

The command above uses two configuration files. The first one is for docker-compose to set up the maestro-quiver-agent. The second one is to set up the SUT. The end of the command specifies which and how many maestro components should be initialized. The testing needs only one agent as the testing itself is managed by Quiver.

After the cluster is set up, the test broker needs to be started. First of all, an instance of a broker is created with configuration as follows:

```
./bin/artemis create --user <userName> \
--password <password> --role amq \
--allow-anonymous <instanceName>
```

The newly created instance is run by the command:

```
<path/to/instanceName>/bin/artemis run
```

Next setup is to run the maestro client:

```
docker run -it -h maestro_client \
-v maestro:/maestro --network=cluster_cluster \
maestroperf/maestro-client:testing-1.5 /bin/bash
```

Within the maestro client, the tests can be configured and run. According to section 3.1, the user needs to set up environment variables before the testing starts. The configuration is as follows:

```
export SOURCE_URL= \
https://github.com/dstuchli/maestro-quiver-agent.git
export SOURCE_BRANCH=master
```

The above configuration uses a fork of the maestro-quiver-agent because the fork is modified to use the Quiver Results Converter. The branch of the git project is also specified.

5.2 Tests

Tests consist of two simple test runs in the environment described above. The APIs chosen to be tested are Python and C++ implementations. Quiver identifies those implementations as `qpid-proton-python` and `qpid-proton-cpp` respectively. To run the quiver tests using Python API, the user must specify the Quiver arrow implementation in the environment variable `TEST_PARAMS` in maestro client:

```
export TEST_PARAMS=qpid-proton-python
```

Maestro uses `FlexibleTest` script to run Quiver tests:

```
maestro-cli exec -s \
/opt/maestro/maestro-cli/scripts/singlepoint/ \
FlexibleTest.groovy
```

Analogically the same configuration is run for C++ API with only difference being the Quiver arrow implementation specification:

```
export TEST_PARAMS=qpid-proton-cpp
```

5.3 Data Collection and Evaluation

Thanks to Quiver Results Converter, the results are visible on `localhost:6500`.

The results for Python implementation taken from web UI:

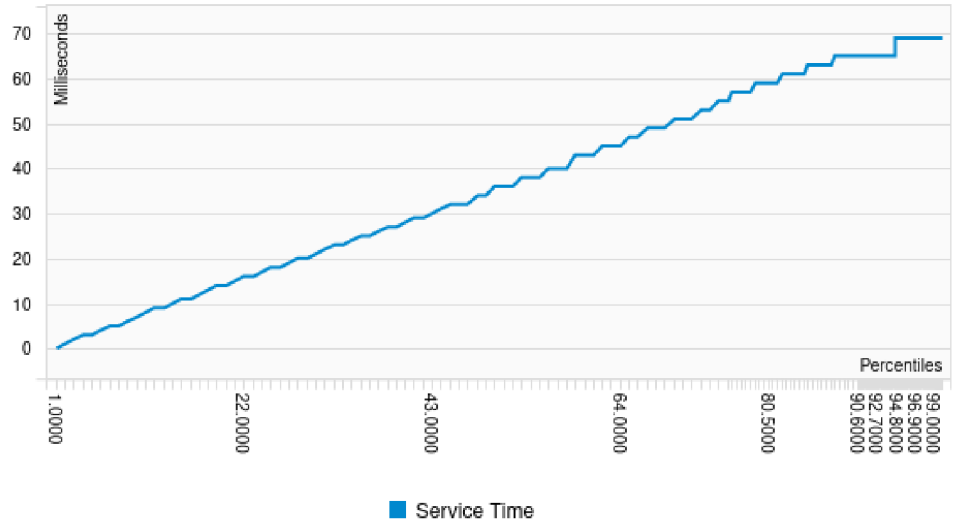


Figure 5.1: Distribution of the Latency Percentiles

Metric	Value
Total Count	1000000
Max Latency	69695 µs
99.99th Percentile	69695 µs
99.9th Percentile	69695 µs
99th Percentile	69695 µs
95th Percentile	69695 µs
90th Percentile	65599 µs
50th Percentile	36895 µs

Figure 5.2: Service Time Statistics

In the charts 5.3 and 5.4 user can see the number of messages sent per time unit.

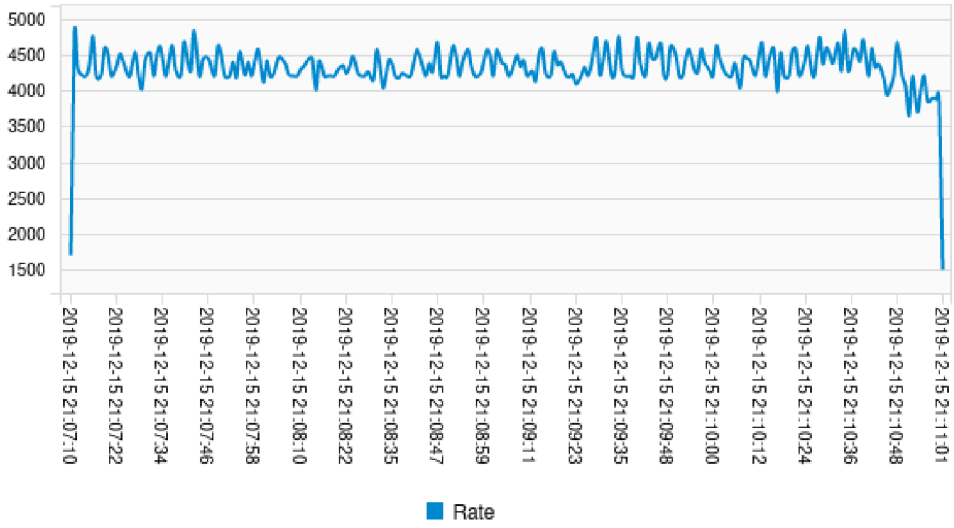


Figure 5.3: Sender Rate Distribution Over Time

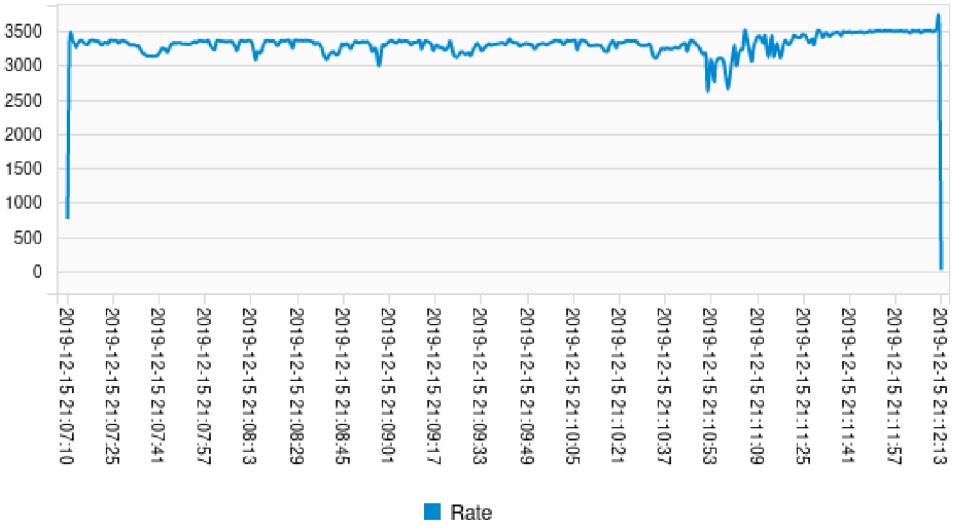


Figure 5.4: Receiver Rate Distribution Over Time

In the tables 5.5 and 5.6 user can see information about the sender and receiver overall statistics.

Metric	Value
Maximum Rate	4834.00 msg/sec
Minimum Rate	1500.00 msg/sec
Mean	4310.34 msg/sec
Geometric Mean	4292.27 msg/sec
Standard Deviation	321.80
Rate Samples	N/A
Rate Error Count	N/A
Skipped Rate Records	N/A

Figure 5.5: Sender Rate Statistics

Metric	Value
Maximum Rate	3511.00 msg/sec
Minimum Rate	12.00 msg/sec
Mean	3289.47 msg/sec
Geometric Mean	3230.10 msg/sec
Standard Deviation	268.61
Rate Samples	N/A
Rate Error Count	N/A
Skipped Rate Records	N/A

Figure 5.6: Receiver Rate Statistics

The results for C++ implementation taken from web UI:

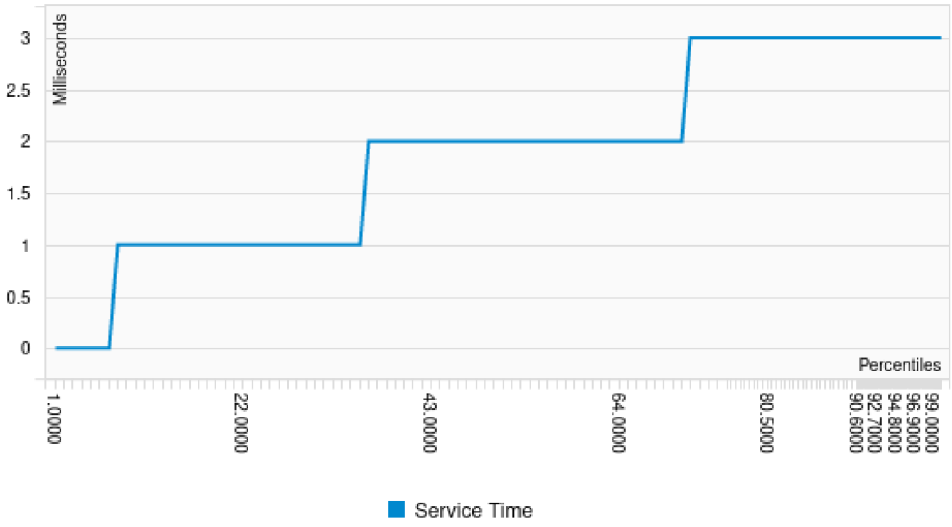


Figure 5.7: Distribution of the Latency Percentiles

Metric	Value
Total Count	1000000
Max Latency	3457 µs
99.99th Percentile	3457 µs
99.9th Percentile	3457 µs
99th Percentile	3457 µs
95th Percentile	3457 µs
90th Percentile	3329 µs
50th Percentile	2433 µs

Figure 5.8: Service Time Statistics

In the charts 5.9 and 5.10 user can see the number of messages sent per time unit.

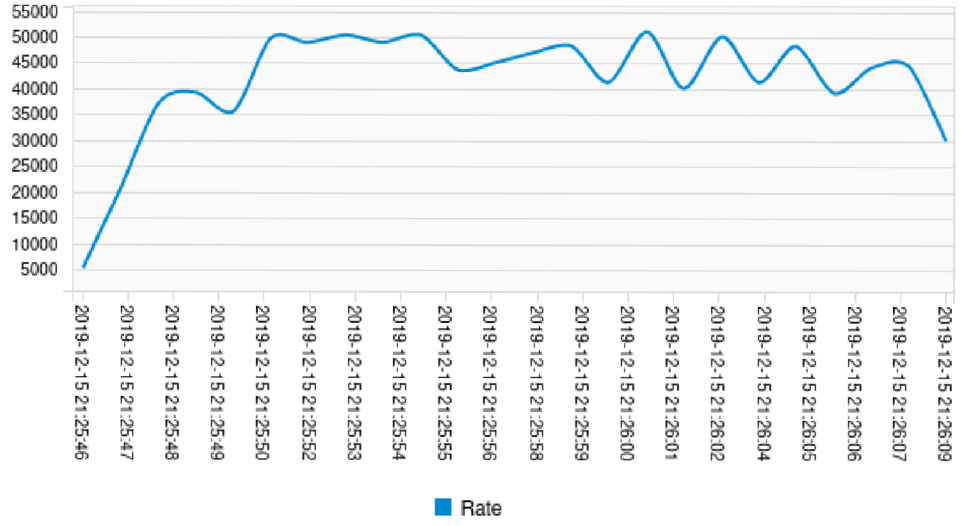


Figure 5.9: Sender Rate Distribution Over Time

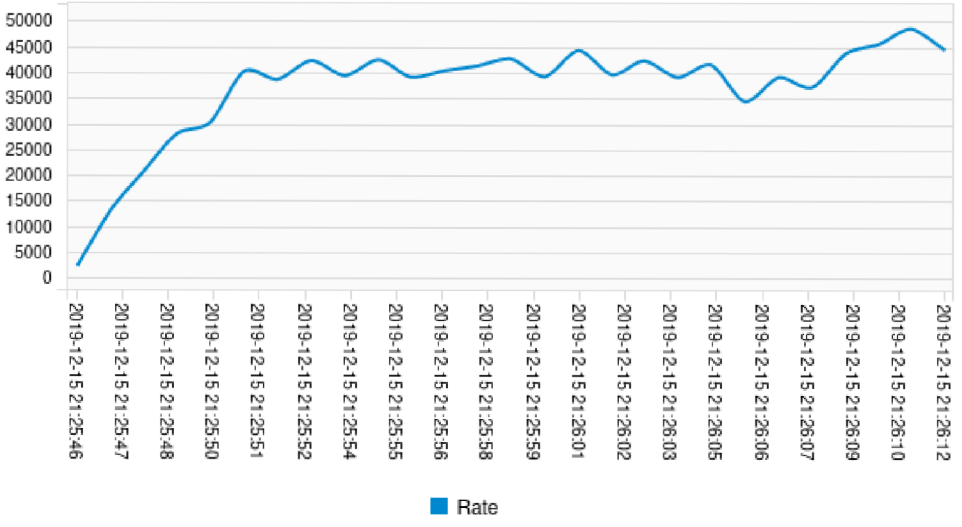


Figure 5.10: Receiver Rate Distribution Over Time

In the tables 5.11 and 5.12 user can see information about the sender and receiver overall statistics.

Metric	Value
Maximum Rate	51100.00 msg/sec
Minimum Rate	5200.00 msg/sec
Mean	41666.67 msg/sec
Geometric Mean	38905.81 msg/sec
Standard Deviation	10685.74
Rate Samples	N/A
Rate Error Count	N/A
Skipped Rate Records	N/A

Figure 5.11: Sender Rate Statistics

Metric	Value
Maximum Rate	48631.00 msg/sec
Minimum Rate	2238.00 msg/sec
Mean	37037.04 msg/sec
Geometric Mean	33582.48 msg/sec
Standard Deviation	10305.95
Rate Samples	N/A
Rate Error Count	N/A
Skipped Rate Records	N/A

Figure 5.12: Receiver Rate Statistics

Result Analysis

From the charts and tables listed above, we can see that the C++ implementation of the messaging client has much better performance. The C++ client has much better latency distribution. Python implementation suffered almost 70-millisecond latency against the C++, which has only 3.5 milliseconds at most. C++ client's sender and receiver rate are almost ten times more significant than Python's. Python client also has bigger oscillations than the C++ client.

The final verdict of this test is that the C++ behaves much better and has much better overall statistics with lower latency, higher message rate, and more stable rate distribution than Python client.

6 Conclusion and Future Work

In this thesis, the basics of software testing were introduced. More emphasis was given to the Performance testing. The process, the types, and the metrics of performance testing were summarized. The Maestro testing tool was described along with another tool called Quiver used for performance testing of messaging clients. Part of the Maestro tool is also maestro-quiver-agent, which is used to run Quiver tests within the Maestro test cluster. The Quiver Results Converter for Quiver data conversion to Maestro Binary Rate Format was described.

The converter solves one of the problems with the maestro-quiver-agent. There are vast differences between the Maestro and Quiver implementation. The consequence is that Maestro cannot read the data from Quiver appropriately. In-depth knowledge of both Maestro and Quiver were needed to solve the issue. The result is a standalone CLI utility that processes the Quiver results and generates files supported by Maestro. Part of the generated files should be `system.properties` which is not generated for now as it is not mandatory to evaluate the results. The maestro-quiver-agent now uses this utility, so it is easier for Maestro to fetch the files and represent the data in web UI.

Part of this thesis was to test two APIs using this converter. The Python and C++ APIs were tested, and the results were evaluated based on the converted data showed via web UI.

The converter helps with automatization of performance testing as engineers from Red Hat can now use only one tool for the testing.

In the future, the converter will receive better file generation as the Maestro has a very sophisticated way to collect the data. An improvement in code efficiency is also an option. The maestro-quiver-agent can be optimized to support running tests with different peer implementations and different protocols then AMQP.

Bibliography

1. STEJSKAL, J. *Performance Testing and Analysis of Qpid Dispatch Router*. 2018. Available also from: `http://www.fit.vutbr.cz/study/DP/DP.php?id=21191`. Master's thesis. Brno University of Technology – Faculty of Information Technology.
2. *Maestro Java* [online]. Piske, O. R. [visited on 2019-02-25]. Available from: `https://github.com/maestro-performance/maestro-java`.
3. *Quiver* [online]. Ross, J. [visited on 2019-03-05]. Available from: `https://github.com/ssorj/quiver`.
4. HEROUT, P. *Testování pro programátory*. České Budějovice: KOPP, 2016. ISBN 978-80-7232-481-1.
5. BOEHM, B. W.; MCCLEAN, R. K.; URFRIG, D. E. IEEE Transactions on Software Engineering 1: Some Experience with Automated Aids to the Design of Large-Scale Reliable Software. In: IEEE, 1975, pp. 125–133.
6. PATTON, R. *Testování softwaru*. Brno: Computer Press, 2002. ISBN 80-7226-636-5.
7. MCCONNELL, S. C. *Code Complete (2nd ed.)* Washington D.C., USA: Microsoft Press, 2004. ISBN 978-0-7356-1967-8.
8. DIJKSTRA, E. W. *Classics in software engineering*. Upper Saddle River, New Jersey, USA: Yourdon Press, 1979. ISBN 0-917072-14-6.
9. *IEEE Standard Glossary of Software Engineering Terminology* [online]. IEEE [visited on 2019-05-10]. ISBN 978-0-7381-0391-4. Available from: `https://ieeexplore.ieee.org/servlet/opac?punumber=2238`.
10. HUTCHESON, M. L. *Software testing Fundamentals: Method and Metrics*. Hoboken, New Jersey, USA: John Wiley & Sons, 2003. ISBN 978-0471430209.
11. OBERKAMPF, W.L.; ROY, C.J. *Verification and Validation in Scientific Computing*. Cambridge, United Kingdom: Cambridge University Press, 2010. ISBN 978-1139491761.

12. *Řízení softwarových projektů* [online]. Bureš, M. [visited on 2019-05-14]. Available from: https://moodle.fel.cvut.cz/pluginfile.php/98180/mod_resource/content/2/RSP_prednaska_2__v3.pdf.

13. [online]. International Software Testing Qualifications Board [visited on 2019-12-15]. Available from: https://glossary.istqb.org/en/search/.

14. DIN, G. *A Performance Test Design Method and its Implementation Patterns for Multi-Services Systems*. 2009. Available also from: https://depositonce.tu-berlin.de/bitstream/11303/2406/2/Dokument_29.pdf. Master's thesis. Technische Universität Berlin - Fakultät Elektrotechnik und Informatik.

15. *Early Performance Testing of Distributed Software Applications* [online]. Denaro, G., Polini, A., and Emmerich, W. [visited on 2019-05-21]. Available from: http://www1.isti.cnr.it/~polini/downloads/wosp04/wosp2004.pdf.

16. GAO, J. Z.; TSAO, H.-S. J.; WU, Y. *Testing and Quality Assurance for Component-Based Software*. Norwood, Massachusetts, USA: Artech House Publishers, 2003. ISBN 1-58053-480-5.

17. *Measuring Component PerformanceUsing A Systematic Approach and Environment* [online]. Gao, J., Ravi, C. S., and Raquel, E. [visited on 2019-05-21]. Available from: https://subs.emis.de/LNI/Proceedings/Proceedings58/GI.Proceedings.58-6.pdf.

18. *Maestro Quiver Agent* [online]. Piske, O. R. [visited on 2019-12-14]. Available from: https://github.com/maestro-performance/maestro-quiver-agent.