BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PERFORMANCE TESTING AND ANALYSIS OF QPID DISPATCH ROUTER

TESTOVÁNÍ A ANALÝZA VÝKONNOSTI QPID DISPATCH ROUTERU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. JAKUB STEJSKAL**
AUTOR PRÁCE

SUPERVISOR — **Ing. TOMÁŠ FIEDOR**
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

Application performance testing has recently become more important during the application development of all kinds. This paper maps the fundamentals of performance testing that are commonly used. It analyze performance testing of components used in Messaging systems, especially Messaging Broker and Qpid-Dispatch. Currently used methods for performance testing of this components are primarily focused on Messaging Broker by Messaging Performance Tool. This paper also describes improvements of Messaging Performance Tool to extend performance testing of Qpid-Dispatch and its capabilities in automatic testing. The final report evaluates the proposed application, the performance of Qpid-Dispatch component and develops ideas for future works.

Abstrakt

Výkonností testování aplikací nabírá v poslední době na důležitosti během vývoje aplikací všeho druhu. Tato práce mapuje základy testování výkonu, které jsou aplikovatelné na libovolné aplikace. Následně analýzuje testování výkonu komponent používaných v Messaging systémech a to konkrétně Messaging Broker a Qpid-Dispatch. Využívané metody testování výkonu je zaměřeno zejména na Messaging Broker pomocí Messaging Performance Tool. Dále je popsáno vylepšení této aplikaci o rozšíření testování systému Qpid-Dispatch a její možnosti při automatizovaném testování. Výsledná zpráva vyhodnocuje navrženou aplikaci, zhodnocuje výkon komponenty Qpid-Dispatch a rozvíjí myšlenky pro další rozšíření.

Keywords

Testing, performance analysis, performance testing, network technologies, router, Qpid-Dispatch

Klíčová slova

Testování, analýza výkonu, testování výkonu, síťové technologie, router, Qpid-Dispatch

Reference

STEJSKAL, Jakub. *Performance Testing and Analysis of Qpid Dispatch Router.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Fiedor

Performance Testing and Analysis of Qpid Dispatch Router

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Tomáš Fiedor. The supplementary information was provided by Ing. Zdeněk Kraus and Otavio Rodolfo Piske from Red Hat, Inc. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jakub Stejskal
April 28, 2018

Acknowledgements

I would like to thank to my supervisors, Ing. Tomáš Fiedor from BUT VUT and Ing. Zdeněk Kraus from Red Hat, Inc. for guiding and important information about performance problems. Also to my colleagues Otavio Rodolfo Piske for his time during introduction and explanation of his Messaging Performance Tool and Dominik Lenoch for Qpid-Dispatch introduction.

Contents

1 Introduction

Good application performance is one of the main goals during the software development. But what makes software performance so important? Software reliability has to be guaranteed by the owner, but with undesirable performance there could be a lot of issues. Thus can badly influence software behavior. And this can cause a significant outflow of the consumers, and even brand destruction, financial damage, or loss of trust. These few reasons should be enough to do a proper performance testing before every software release, especially for large projects where industries guarantee certain level of software behavior and they would not be able to assure it with insufficient performance testing. Great emphasis on software performance is, in particular, in space programs, medical facilities, army systems, or energy distribution systems. In these fields it is necessary to ensure proper application behavior for a long time under high load and without any unexpected behavior such as high response time, frequent delays, or timeouts, because every failure is paid dearly.

Nowadays every developer should try to use well established frameworks which can make theirs work easier. Frameworks handle complex underlying issues such as security, performance, and code clarity. This way developers can invest more time in the actual functionality and meet the application requirements, since frameworks are usually optimized for one particular job. In the past every developer had to spent significant portion of development time tuning performance which led to spending more time and money for software development. But not everyone has enough knowledge of performance testing and this makes performance analysis and optimization even more difficult. This leads to a need for specialized performance tools which can provide more sophisticated information, however, actually useful tools are usually proprietary and/or too expensive.

A very important part of the performance analysis is the right choice of *key performance indicators* (KPIs) [18] and effective interpretation of the results. The right choice of KPIs allows faster detection of performance problems and help developers with fixes and meeting the *performance standards* [18] set up by application owner or customer in time before the release.

In general an application performance is important. However, smooth network application or hardware performance became much more demanded nowadays, since most of the communication is via the Internet. Obviously when you make a payment in your internet banking you definitely want to have a stable connection to your bank's website without any delay. Network stability is significantly influenced by network components like routers and switches and hence their performance should be under utmost case. We refer to network performance testing as measurement of network service quality which is directly influenced by *bandwidth, throughput, latency*, etc.

For performance testing of particular network messaging system developed by *Red Hat Inc.* there is an existing solution —Messaging Performance Tool (MPT) [20]. MPT is specialized for the performance testing of *Message Broker* (Broker) [21] — network application

level software cooperating with *Qpid-dispatch service* [22] in the network as the message distributor. Unfortunately, the current version of MPT does not support performance testing of enough component like the Router component, Qpid-dispatch. In this work we will focus on this particular short coming and develop a worthy solution allowing proper performance testing of the Qpid-dispatch service.

This thesis is structured as follows. First, we define fundamentals of performance testing in Chapter 2. The rest of the thesis focuses on performance testing and analysis of Qpid-dispatch, an application level router designed by Red Hat Inc. Qpid-dispatch performance testing is based on MPT described in Chapter 3. Description includes *measurement process* and *measured data description and evaluation*.The main goal of the thesis is to analyze MPT and design module for the Qpid-dispatch performance testing as described in Chapter 4 together with used protocols and *Automatic Topology Generator* for semi-automated network generation and deployment. Used technologies, tools and implementation processes of each component are described in Chapter 5. The most important part of the thesis is Chapter 6, containing the data gathering from routers located in different types of topology, data evaluation and representation which leads to conclusion about performance of Qpid-dispatch. Finally, Chapter 8 summarizes the thesis and proposes ideas for future use of developed tool.

2 Fundamentals of Software Performance Testing

Usual goal of performance testing is to ensure that the application runs reasonably fast enough to keep the attention of users, even with unexpected amount of clients using the application at the same time. But why is it so important to have the application optimized for the best speed? Simply, when your application have slow response, long load time or bad scalability, the first website which user will visit afterwards will be web of your competitor. That is the reason why speed is currently one of the most significant performance factor of common performance problems. This chapter summarizes the fundamentals of performance testing which includes common performance processes, issues, and metrics, based on knowledge available in [18, 17, 12, 2].

2.1 Performance Testing Process

The main goal of the performance testing is to ensure the following application attributes [13]:

Reliability and Stability — the ability of software to perform its functions in system environment under some system load for acceptable[1] period of time,

Scalability — the ability of software to behave properly under various types of system load and handle increasing amount of workload (such as network traffic, server load, data transfer, etc.) which need new hardware for cluster expand,

Processing time and Speed — the ability of software to react quickly without low response time during any acceptable system load,

Availability — the ability of software to make all of its functions available during any acceptable system load. The ability of software, deployed in cluster, to provide all functions during node crash is called High Availability.

Similarly to software development process, performance testing process consist of usual engineering steps ranging from requirements definition to data evaluation. These steps also includes design, implementation, and execution of performance tests with data collection. The graphical representation of the performance testing process is depicted in the Figure 2.1.

[1] During software development there is a document with Software Requirements Specification which specifies software metrics, including performance.

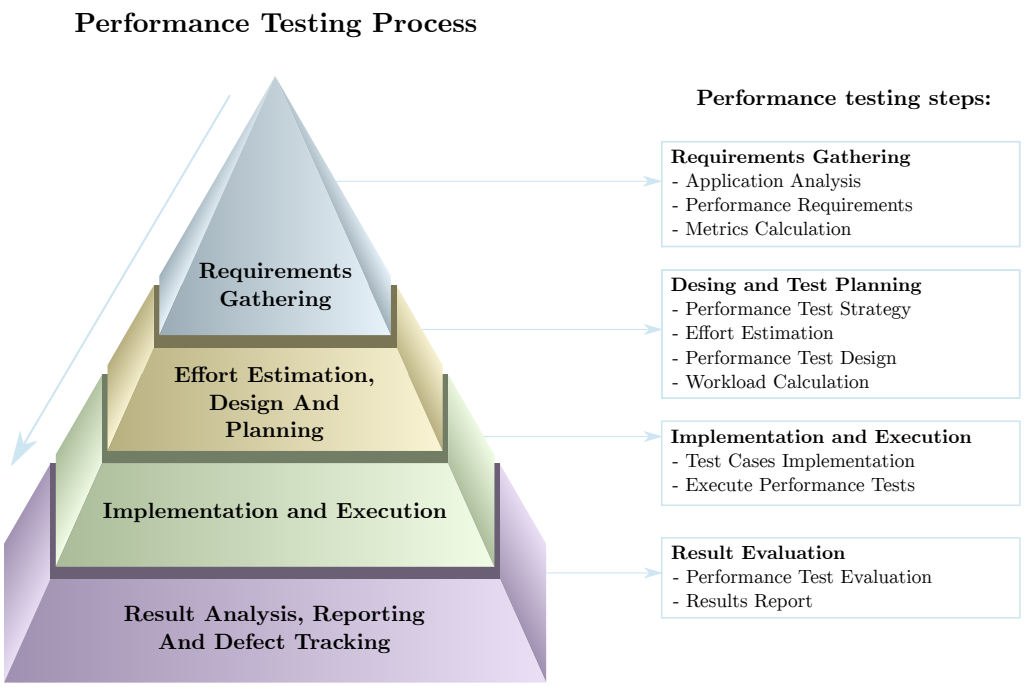


Figure 2.1: Performance Testing Process with the four most important parts and theirs individual steps based on [23].

In the Figure 2.1 you can see scheme of performance testing process where each level represent time for each step. Lower level refer to more time spend on that step.

The first step of performance testing process is the selection of *performance requirements* for the application. In this step, testing engineer has to analyze *software under test - SUT*, suitable performance metrics, that will model the application performance, and set performance requirements usually with customer and project manager. The result should include answers to questions such as:

- How many end users will the application need to handle at release, after 6 months or in 1 year ?
- Where will these users be physically located, and how will they connect to the application?
- How many end users will be concurrently connected in average at release, after 6 months and 1 year?

Based on answer to these studies, the engineer should be able to select important key performance indicators for performance test cases. Some of these indicators may be *response time*, *stability*, *scalability*, or *speed*. However, there is huge amount of possible indicators so it is necessary to properly analyze the whole application and also take into consideration another needs like an error rate, system resources, etc. Result of this phase should be

a binding document with all performance requirements to be tested and, in case of detected performance degradation, such defect must be fixed with reference to this document.

The next step is to define the *performance testing strategy*, corresponding to planning and design. It is extremely important to allocate enough time for SUT testing effectively, because, as it was mentioned in Chapter 1, performance testing is not an easy task and detecting all of the possible issues of tested components is very time consuming process. Every plan should take into account the following considerations:

Prepare the test environment — this step include choosing right hardware for testing, then installing the necessary software for running load injectors, tested components, etc., and other equipment depend on application purpose such as routers, switchers, mobile devices, etc.

Provide sufficient workload injectors — preparing the workload injector may take few days; we usually requires a few workstations or servers to simulate real traffic.

Identify and implement use cases — this include identification of important parts of the system which may have an impact on performance; time needed for each use case may be different because some use cases can be simple such as navigating to a web application home page, but some may be complex such as filtering specific communication.

Instrument the test environment — install and configure the monitoring software on the test environment.

Deal with detected problems — test can detect significant performance issues, but their investigation and fix may take a long time. After fix the retest of issue is needed.

While this process seems trivial, the opposite is true, in especially in case of network applications. Most of performance issues manifest at big workloads or high number of users, e.g. when million users are sending requests to the network device at the same time it could lead to an unacceptable device crash. Workload injectors are designated to simulate real user activity, and allows automatic analysis of performance behavior for tested application or device. Depending on the used technology, there can be a limit on the number of virtual users that can be generated by a single injector. These automated workload injectors are necessary for effective performance testing.

After describing the plan we implement and execute proposed test cases. Environment and workload injectors are ready for execution, so last step before the testing itself is the implementation of tests. Thanks to the careful planing, engineers should have enough time to implement test cases with reference to proposed design.

Final step of performance testing process is results evaluation. Output of this step is usually technical report with all selected performance key indicators, used workload and Collected Data Format for each test case. Then follows the data evaluation with thorough analysis of degradation localization. Additionally, the report usually contains syntactical graphs which display performance metrics along the duration of test execution.

2.2 Performance Issues

Performance issue is a common label for an unexpected application or device behavior which affects its performance. Usually, those issues are hard to detect because they manifest only under certain circumstances such as high load or long application run time. In the network

applications there are several particular issues that are more frequently occurring than others. In following, I will describe selected issues in more detail.

Performance Degradation

Unclean code usually leads to inefficient algorithms, application deadlocks, or memory leaks, which all can eventually cause the performance degradation. The problem is that these issues are usually detected only during the long run time of application or inability of an application to handle high load. For this kind of issues there is a performance testing method called *soak testing* [9, 15] which is described in Section 2.3. Soak test is intended to identify problems that may appear only after long period of application run-time[2], hence its necessary to run this type of tests during application development. The network applications are usually need to be available for 24 hours per day. The duration of a soak test should have some correlation to the operational mode of the system under test. Following scenarios may represent performance issues detectable by soak tests:

- a constant degradation in response time, when the system is run over the time,
- any degradation in system resources that are not apparent during short runs but will surface during long run time such as free disk space, or memory consumption
- a periodical process that may affect the performance of the system, but can be detected only during long run time as backup process, exporting of data to a 3rd party system, etc.,
- development of new features for already exists components.

Response Time

Response time is how long it takes system to accept, evaluate, and respond to the user for his request e.g. HTTP request for particular website. Different actions and requests can have significantly different response time and with that provide different load on the system. For example retrieving document from web-server by its ID is considerably faster than searching for the same document by keywords. Response time is mostly measured during the *load test* [15] of the application. Well designed test should consider different types of load on the system, various kind of requests, and different number of connected end-users at the same time. For user based systems we usually consider 3 threshold for the response time values:

0.1 second — this represent an ideal response time for the application, because user feels that system is reacting instantly and does not notice any interruptions.

1 second — this is the highest acceptable response time when user still does not feel any interruptions, but can feel a little delay; this still represent no bad impact on the user experience.

10 second — this is the limit after which response time become unacceptable and user will probably stop using your application.

[2] Soak Test - refer to HW testing method during which engineers soak device into water and check for bubble leaks.

However response time limits for non-human interactive system are more strict. They could acquire values in milliseconds or less.

[[Prvni iterace]]

Traffic Spikes

As *traffic spike* [17, 6] we can understand the sudden surge in demand from users. Typically doubling or multiplying of traffic level in short period of time. In real network, spikes are result of high workload, e.g. caused by higher amount of users trying to concurrently use the service over the network. For example we can experience sudden traffic spike in response time after publishing new popular viral context on video servers, start of sales events, reservation of limited amount of tickets or subject registration at university. Scheduled automatic backup or system upgrade for whole company during early morning hours can cause traffic spikes.

Traffic spikes can lead to inappropriate system behavior such as *long response time*, *bad throughput*, and *limited concurrency*. To prevent the impact of traffic spikes on system performance, it is necessary to do sophisticated infrastructure monitoring and network load analysis, in order to distinguish between normal traffic and attack on the system. Suitable methods for testing of spikes is one of variant of *stress testing* [15] and it is described in Section 2.3 in more details. Network system should offer load balancing, thus it should be able to redirect traffic to another node with same service in case of high load which can cause performance issues due inappropriate resource usage.

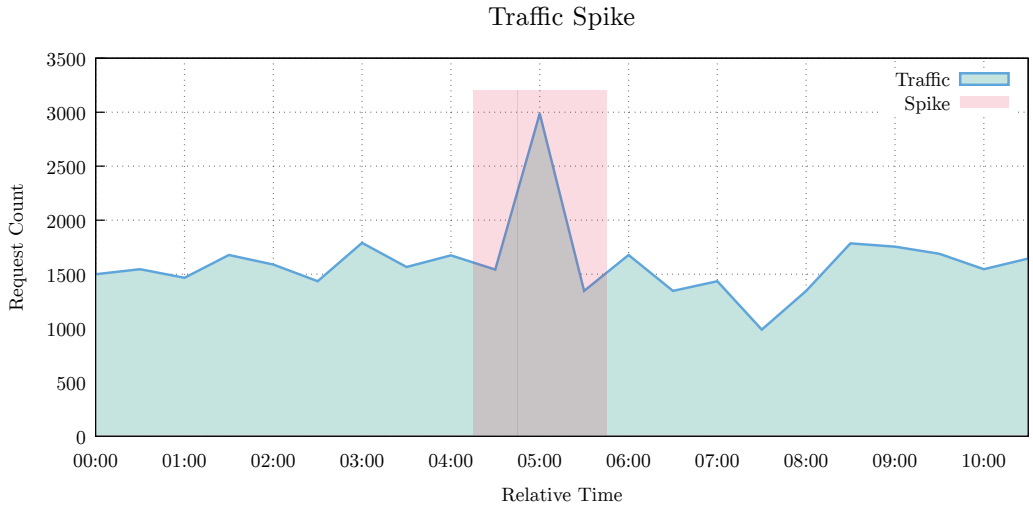


Figure 2.2: The graph shows amount of concurrent sessions depending on time. During to network traffic monitoring the traffic spike occurring after 5 hours from test start.

2.3 Types of Performance Testing

For performance testing there are many types of suitable test methods. Which test you should use is determined by the nature of the system, testing requirements or how much time we have left for the performance testing. The following terms are generally well known and used in practice and each of them characterizes category or suite of the tests:

- **Testing methods** — load testing, stress testing, endurance testing, [[**scalability, volume, recovery**]]
- **Testing approaches** — smoke testing, regression testing, benchmark testing

Their description is based on knowledge available in [5, 9, 18, 2].

Load Testing

Finding maximal load is a testing method which studies how the system behaves during different types of workload within acceptable time range. Basically it simulates the real-world load. During the load test we mainly focus on metric response time of the system for requests. Requests are generated by users or another systems communicating with the SUT. Main goal is to determine if the system can handle required workload according to performance requirements. Load test is designed to measure the response time of system transactions under normal or peak workload. When the response time of the system dramatically increases or becomes unstable, we conclude that system reached its maximum operating capacity. After successful testing, we should mark the workload requirements as fulfilling or analyze Collected Data Format and report issues to the developers. In the Figure 2.3 you can see the graph of load test showing workload of raising requests to the web server at the same time where the system response time does not exceed 3.5 seconds.

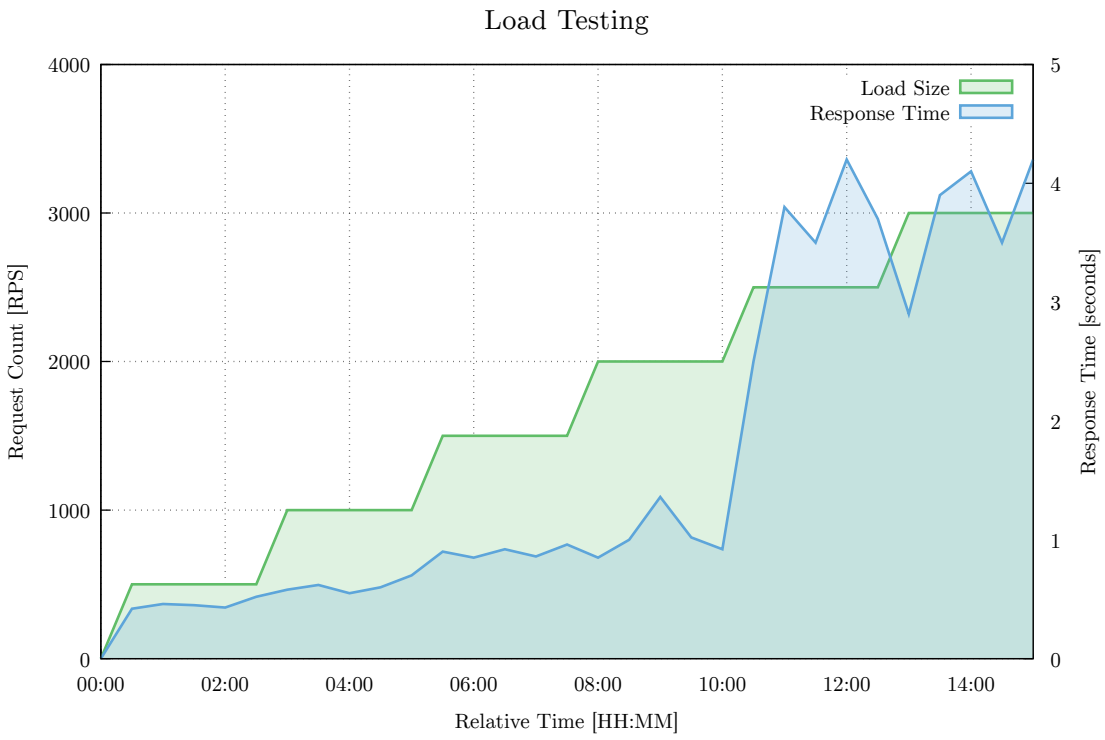


Figure 2.3: Response time of the system during the load testing depended on load size.

The following lists show common scenarios for load testing:

- The system interact with multiple users at same time.
- The system tracking communication and analyze it.

- Web services and information systems.

Typical system issues covered by load testing:

- Concurrent users connections can eventually result into slow response time or system crash.
- Network systems without redundancy connections can shutdown whole network under normal defined workload.
- Data availability during multiple session to data server.
- Connection rejection (timeout).

Stress Testing

Stress testing is the specific type of load testing, where we do not measure normal workload, but focus on unexpected workloads or traffic spikes. The main purpose is to study how the system behaves in extreme conditions such as enormous number of concurrent requests, using a server with much less memory or a weaker CPU, and analyze the system performance threshold. Its very useful to know performance threshold in order to know the difference between performance under normal workload and performance threshold. The following enumeration lists common stress test scenarios:

- Monitor the system behavior with over maximum of users logged in at the same time.
- All user performing critical operations at the same time.
- All users accessing the same file at the same time.
- Hardware issues such as server in cluster down.

Typical issues, which are covered by stress testing:

- Sudden performance degradation.
- System will recover after stress test (system is operational after test).
- System does not crash during stress test.
- All subsystems such as database, load balancer, etc. remains operational.

When engineers finish stress testing and found the limits of the system, they also can test the system recovery after crash during finding of the system limits.

In the Figure 2.4 is recorded stress testing with raising load and response time. Everything is fine until the amount of requests exceed 3000 requests per second. With higher load there comes performance issues which leads to unexpected rise of the response time.

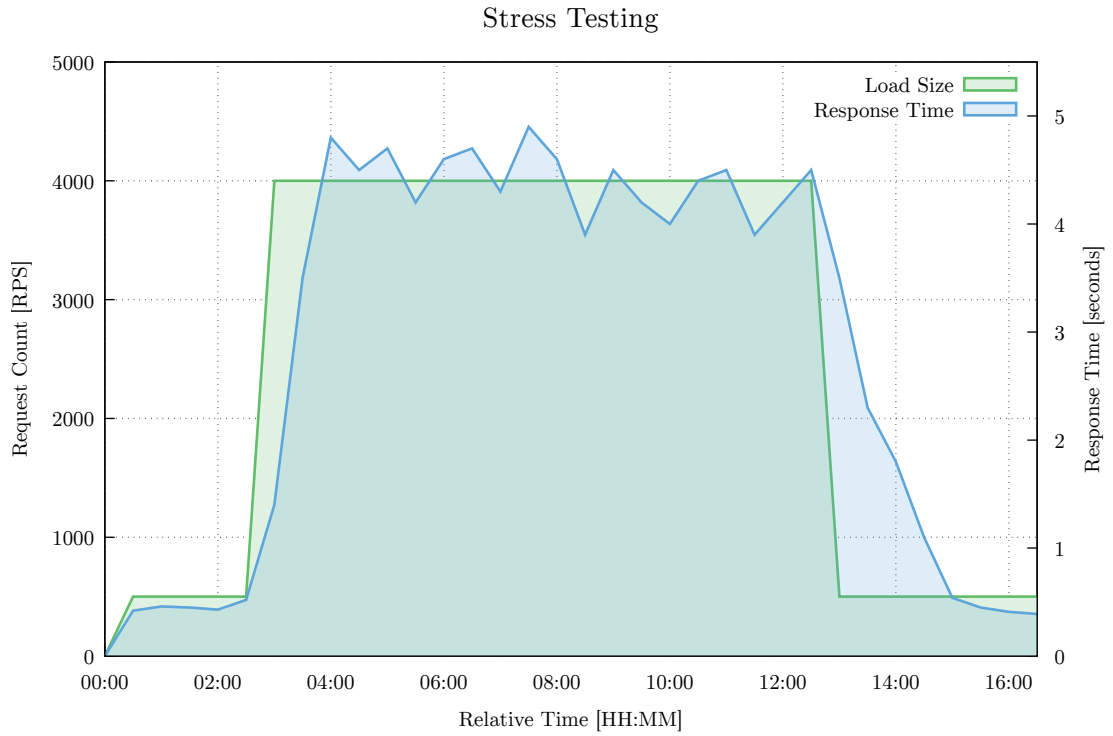


Figure 2.4: Stress testing diagram capturing dependency of response time on amount of requests.

Endurance Testing

Endurance, or stability/soak testing refer to the method, that tries to identify problems, that may appear only after the extended period of time e.g. The system could seems stable for one week, but after some longer period, problems such as memory leaks or not enough disk space can appear. Soak tests mainly focuses on measuring of memory as performance metric. The following are common issues found by soak test:

- Serious memory leaks that can eventually result into the system crash.
- Improperly closed database connections that could starve the system.
- Improperly closed connections between system layers that could stall any of the system modules.
- Step-wise degradation that could lead to high response time and the system becomes inefficient.

Typical scenarios for use soak testing:

- Developed system uses multiple database connections.
- There is a chance for inappropriately allocated memory, and memory free.
- Disk space limitation for store logs or other data.

This sort of test needs to use appropriate monitoring system to achieve high efficiency. Problems detected by soak tests are typically manifested by gradual system slowdown in response time or as a sudden lost of system availability.

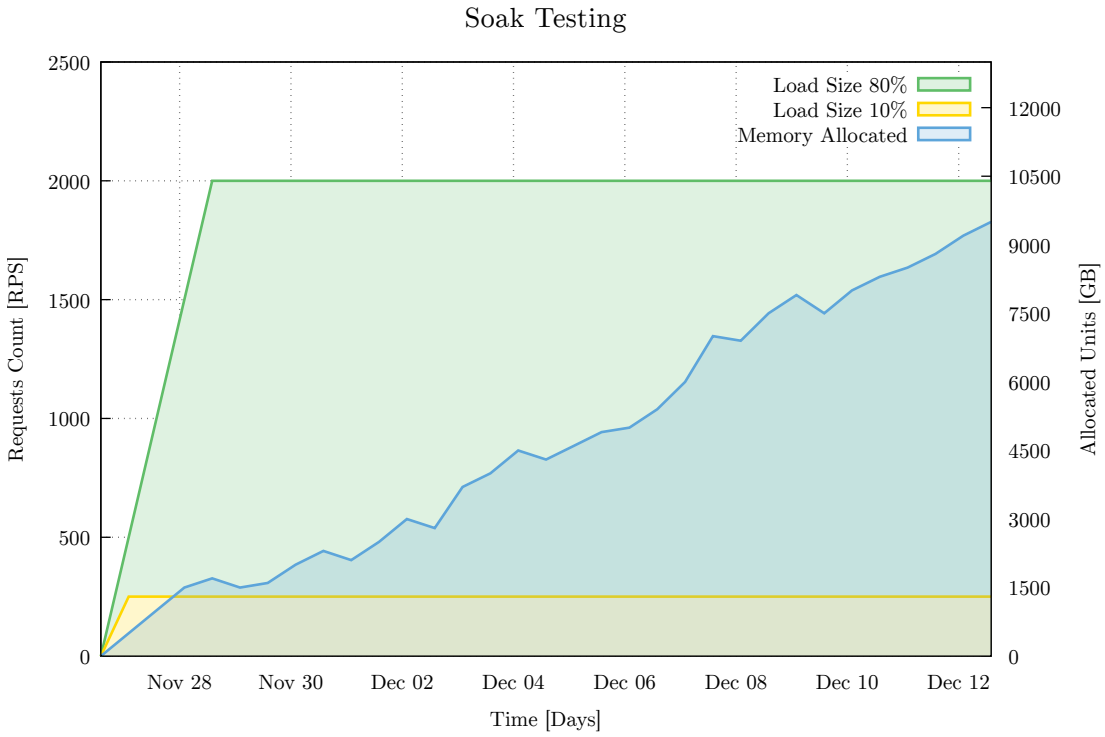


Figure 2.5: Soak testing with memory usage dependent on time.

In the Figure 2.5 you can see raising memory usage after period of time. The SUT can handle requests but as time goes by memory usage is too high that the SUT will crash. This may be caused by memory leak or an inappropriate algorithm use.

Smoke Testing

Smoke testing approach is inspired by similar hardware technique, when engineers checks for presource of the smoke from the device after turning the power on. Basically, its similar for software, since main goal of smoke test is to test basic functionality of the system and guarantee that the system is ready for build. However, smoke tests are testing the functionality on a surface level, so it may not be enough for deep testing of basic system functions. When smoke tests fail, the system is tagged as unstable, because it cannot ensure its basic functionality and it is not tested anymore until the smoke test pass. Smoke test are designed to uncover obvious errors which saves time, money and effort of the engineers. These tests should be used with every new build, since new features could harm previous system functionality. The following lists show common scenarios for smoke testing:

- New system's build or version is ready for further testing or productilization.

Typical system issues covered by smoke testing testing:

- System without main functionality is useless, because test coverage of functionality is low.
- Main functionality can result into system crash.

Smoke testing is not typical performance testing approach, but it can be used for initial load test for check if system can be started.

Regression Testing

Whenever engineers develop new feature and want to update the previous build it has to pass the *regression tests*[3] [4]. Regression tests are designed to test functionality of the latest build updated with new feature. The main objective is to determine, if new feature affects already functional parts of the system. This type of tests is very important, because engineers do not always realize, which parts of the system will be indirectly affected. During regression testing, new test cases are not created, but previous test cases are automatic re-executed and analyzed. Typical scenarios for regression testing:

- New feature of system is ready for use.

Common issues covered by regression testing:

- New feature could adversely affect already working components of the system.

Benchmark Testing

Benchmark testing[3] is approach, which collects performance data during the system run on different hardware machines [16]. Collected Data Format have significant value when we want smooth run of the system on an older hardware, hence we can discover performance issues under normal load. However, the system does not run smoothly on prepared hardware, only options is to run benchmark tests on different machines with different hardware and under different load.

- Can identify minimal requirements for HW, metrics, etc.
- Can validate supported HW configuration.

2.4 Performance Metrics

During the performance testing we can monitor a lot of metrics, which can have different importance based on the system's purpose. The following lists the most common metrics that are monitored during the performance testing of all applications, no matter of developing language.

In the testing systems, performance metrics are collected during long process of collecting, analyzing and reporting information regarding performance of whole system or individual component. This process can be different for each metric, since each metric needs different type of the system analysis.

How to Measure

Performance measurement process can be divided into several steps. Metrics are usually measured after a warm-up period of time after the commencement of traffic, because it takes a while for workload to stabilize. Stabilized workload is necessary for measurements because unstable workload can negatively affect the measurement results. In the Figure 2.6 you can see workload phases with marked part for the performance testing.

[3] Approach for test suits, where are used other methods like Load testing, Stress testing, etc.

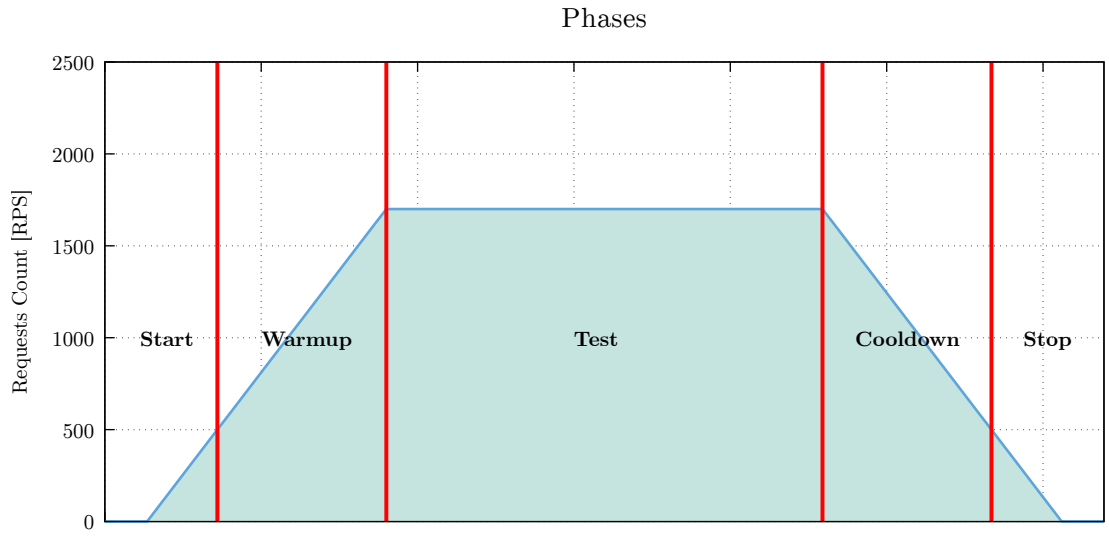


Figure 2.6: Load phases of performance measurement process.

Workload during testing does not have to be on the same value during whole testing. In particular, load testing finds the highest load during which the system can work properly. This limit is find by raising the load and monitor the system as it is shown in the Figure 2.3.

2.4.1 Throughput

Throughput is a metric, which refers to the number of requests per second that the system can handle. *Network throughput* is the rate of successful message deliveries over a communication channel. Throughput is measured by load testing; suitable strategy for measuring throughput is to continuously raise the load until response takes longer that acceptable threshold. [[**neco dopsat**]]

2.4.2 Response Time and Latency

Response time as issue was already mentioned in Subsection 2.2; response time as metric is consists of two parts which are *latency* and *service time*.

Service Time

Service time is the time it takes system to evaluate and send response to user request. In particular, when user sends request for a web page to a server, it takes the server time to evaluate the request and send proper response back to the user, this is the service time. Measurement can be performed easily using stopwatch which starts at receive of request and stops after the response is send. Service time can be affected by any item which leads to a performance degradation as described in Subsection 2.2.

Latency

The second part of the response time is latency [8, 7], which represent a delay between sending the request on the client side and receiving it for evaluation on the server side.

Hence latency is the common problem in the network systems such as data center, web server, etc., because request/response needs to travel over the physical medium between the client and the server. Client and server could be located on different continents, thus the message have to travel long distance and latency is increasing.

Round Trip Time

Round-trip time (RTT) is time that it takes for a signal to be sent with time it takes for an acknowledgement of that signal to be received. In network, the RTT is one of the several factors that affects the signal latency. Basically, RTT is depends on the distance between sender and receiver, because that distance the signals must be travel by.

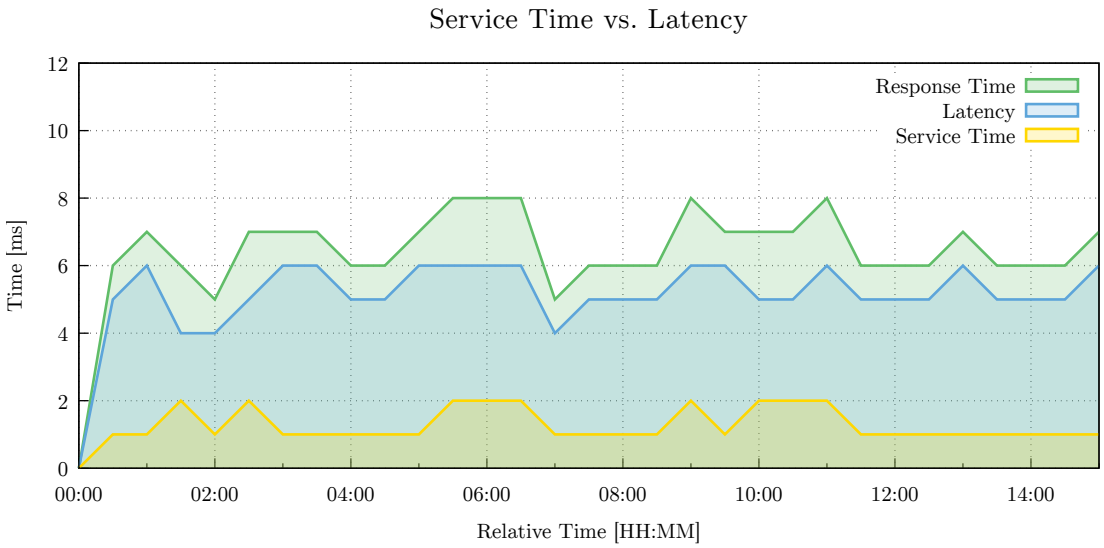


Figure 2.7: Diagram capturing diference between latency and response time.

In the Figure 2.7 you can see response time and both it parts: latency and service time. Service time is usually smaller than latency since latency depends on distance. When you add service time value and latency value you will get response time at certain time.

Average and Percentile Response Time

There are two common ways of measuring the response time [14]: Average (mean) response time is calculated as the sum of all measured times divided by the count of users requests. While this seems trivial, in many times, the average response time does not actually reflect the real response time of the system. How is that possible? In reality, most application have few heavy outliers such as very slow transactions. In the Figure 2.8 you can see few slow transactions which drag the average of the response time to the right. This naturally leads to an inaccurate specification of response time.

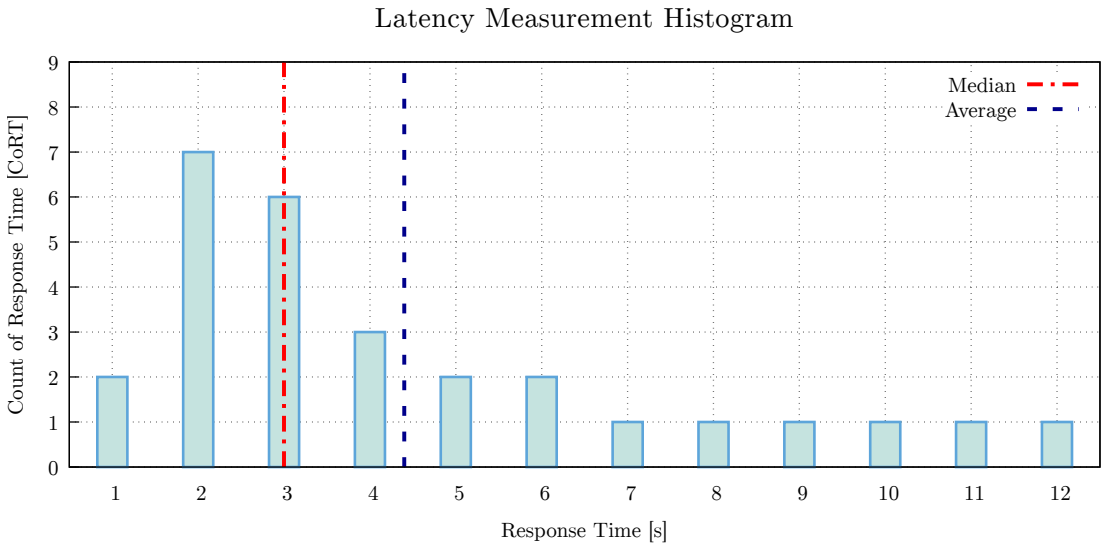


Figure 2.8: Transactions response time with calculated average and median of response time.

In the Figure 2.8 the average represent inaccurate response time, which is higher than real one.

The better solution how to determine the actual response time is Percentile. The percentile is statistic method, which cut measured ordered values into hundredths and then characterize the value below which a given percentage of measurements in a group of particular measurements falls. In the Figure 2.8 you can see the *median* value, which reflects more realistic value of the system response time. Median value is same such as the 50th percentile. In this case, there is no problem, because user will expect slower response time than it has.

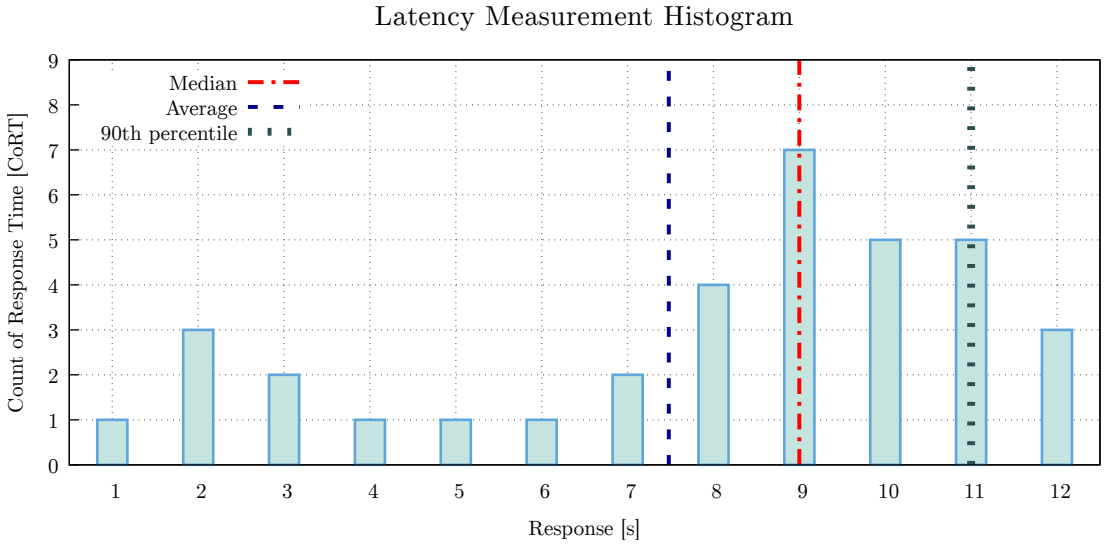


Figure 2.9: Transactions response time with calculated average and median of response time.

The Figure 2.9 shows different situation. Average represent inaccurate response time, which says, that STU is faster than is in reality. Average response time seems better than median, which reflects the expectation of faster system response time than it has. In real systems, we usually use values of the 90th percentile and the 99th percentile. 90th percentile mean, that there is only 10 % transactions slower then marked response time. In the Figure 2.9, a considerable percentage of transactions are very fast (first 50 percent), while the bulk of transactions are several times slower. Thus, calculated percentile gets more realistic values than average response time.

2.4.3 Resource Usage

Applications running at servers with long run-time competes over a limited amount of resource available for use. Thus makes resource usage another important metric, which needs to be monitored since not enough resources could shut down the whole system. Main resources for monitoring and utilization are:

CPU usage — inappropriate usage of CPU could lead to performance degradation, because low priority processes may occupy CPU ahead of the higher priority processes. CPU usage is structuring into system usage and user usage. High system usage can cause problems or bottlenecks.

Memory usage — full consumption of memory could cause performance degradation.

Disk space — for example when using storage disk as a database, there should be preventive measures to backup the data and free up disk space.

Operating System limits — system's memory, and CPU capabilities.

2.4.4 Error Rate

Error Rate is a metric, which commonly occurs in the network systems, especially under high load. During the communication between client and server there could be error caused by another network device (router, switch, etc.) or signal disruption of the data during the transfer. The Error Rate is the mathematical calculation that produces a percentage of problem requests compared to all requests. In the ideal system, there should be zero network error present, however, in reality is in-feasible. This usually leads to a performance degradation and low throughput, because damaged data need to be resent. Error rate is a significant metric because it tells engineers how many requests failed at a particular point in time of performance testing. This metric is more evident when you can see the percentage of problem strongly increasing, hence you can detect problem easily.

[[Druha iterace]]

3 Messaging Performance Tool

Performance of *Message-Oriented Middleware* (MOM) [11] is one of the most critical elements of quality assurance for enterprise integration systems. There are multiple messaging components developed in the Red Hat company such as messaging clients, Message Broker, Message Router (Qpid-dispatch service) or stream-like message distributions tools—Kafka. All of these software needs performance testing to ensure quality standards of MOM. Note that we will shorten term the messaging client to just client in this thesis.

A Message Broker is an example of MOM. Its main purpose is to receive, store and distribute messages, which are sent and received by clients. Users choose MOM for message distribution to reduce the development time and cost of their own solution. Another benefit of using specialized MOM is robustness and guaranteed performance. The performance capabilities of a MOM are critical attributes to its users, because being able to handle a large amount of transactions in a timely manner is a key characteristic of MOM. Good example are automated systems, where components communicates with each other by command exchange. Amount of exchanged commands is heavily dependent on the system size and since we want to get the results as soon as possible we need to ensure smooth and quick message exchange.

Maestro [20] is a testing system designed for testing the performance of MOM. The Maestro is deployed as a cluster system on several machines. A typical deployment consist of one node for Maestro Broker, one or more for Senders, one or more for Receivers and the SUT. The architecture of Maestro system, depicted in the Figure 3.1, consists of the following components:

Maestro Broker — can be any *Message Queuing Telemetry Transport*[1] *(MQTT)* capable broker with several topics. The topic is queue with name where other messaging services can listen the traffic. This component takes care about distribution of control messages between other cluster components such as Maestro Clients and MPT Backend.

Meastro Clients — this component contains the client API as well as the test scripts for each test case. Moreover it contains a sub-component called Reporter which interprets the test data to user in form of web data visualization.

MPT Back-end — consists of sender, receiver and inspector parts. Sender, and receiver ensure message sending to the SUT and receiving from SUT. Inspector monitors workload over the SUT and reports collected performance metrics to the data server. Maestro currently has two backends:

[1] MQTT — http://mqtt.org/

- **Java** — used for JMS-based[2] testing, including *Advanced Message Queuing Protocol (AMQP)* [19], OpenWire and Core protocols.
- **C** — used for AMQP and *Streaming Text Oriented Messaging Protocol*[3] *(STOMP)* protocol testing.

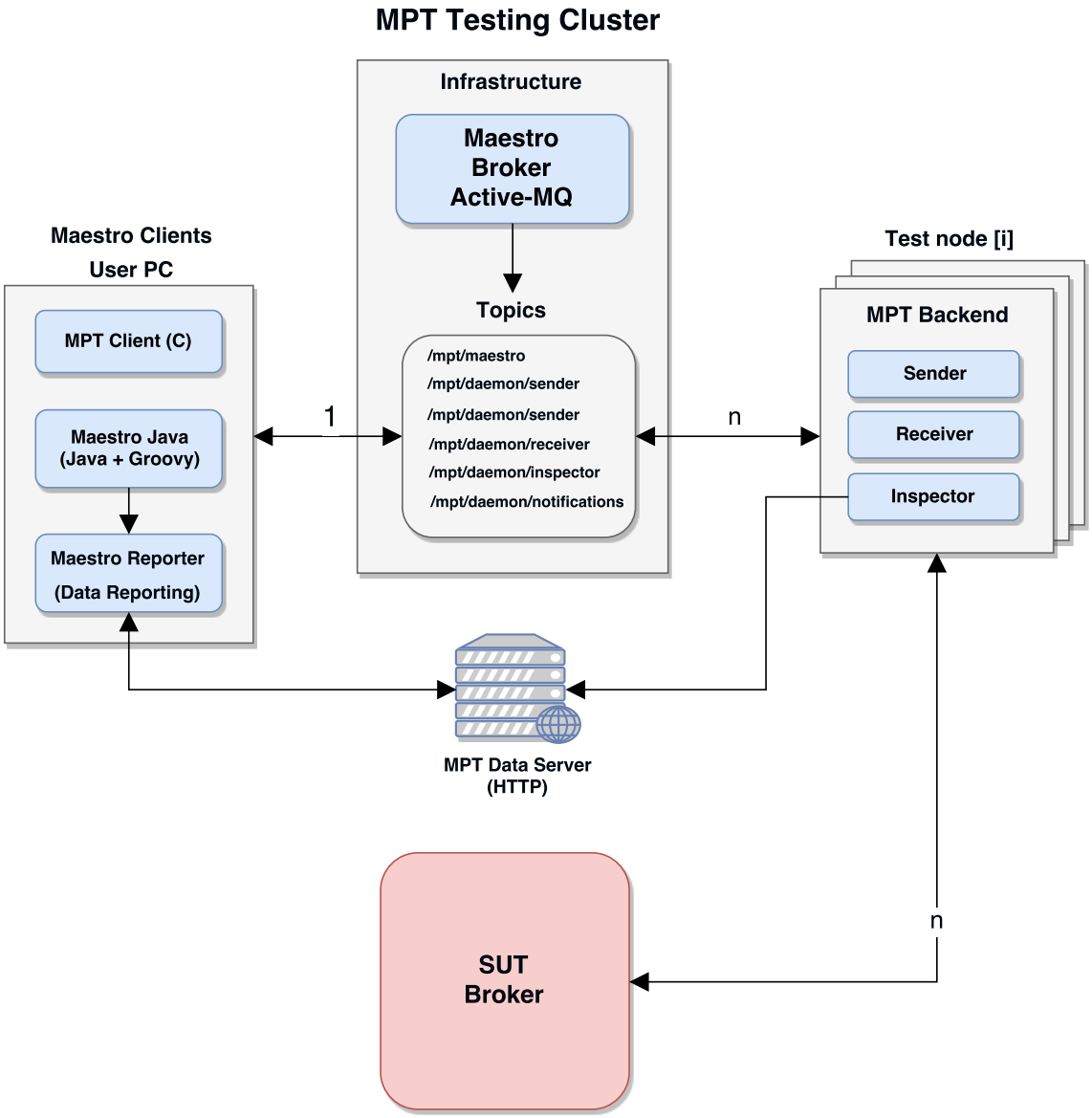


Figure 3.1: Architecture of the Maestro. [[rozepsat]]

[2] JMS — Java Message Service
[3] STOMP — https://stomp.github.io/

3.1 Test Case Scenario

The test is basically generation of huge amount of messages and send them to SUT and then receive them. Configuration of each test case is specified by several options defined in Groovy[4] script which influences the test behavior with the following elements:

- **message size** — size of the generated test message in bytes,
- **number of connected clients** — count of senders and receivers connected to the SUT,
- **test duration (time or load)** — end condition of each test; can be specified by time, limit or message count,
- **message rate** — the desired rate that the system should try to maintain through the test (set 0 for unbounded rate).

The test script is also responsible for start and stop the test. Moreover the test case can be extended by so called test profile. The script will then also be responsible for increasing or decreasing the workload on the SUT during the test scenario. This load can be modified by increasing either the target rate or the number of parallel connections. With multiple combinations of these options we can create a lot of test cases with different loads for the SUT and thus achieve a broad coverage of testing. Every test produces its own logs which are processed by the reporting sub-component on the client side and used for monitoring the metrics. Maestro Reporter produce data visualizations, such as test overview and charts (rate based on time, latency over the test), from these logs.

3.2 Communication Between Components

The actual communication between components during the test cases is realized using the Maestro Protocol — a binary protocol implemented on top of MessagePack[5]. For the message exchange between nodes it currently uses MQTT protocol (version 3.1.1) and for sending the testing data to data server it uses HTTP protocol (version 1.1). The messages exchanged between the peers of testing cluster are called notes.

Each note has a specific format consisting of three parts. First is the *Type* which is short integer that identifies the purpose of the note, it is one of the following values:

- **Request (0)** — a note sent by a controller node to the test peers,
- **Response (1)** — a note sent by a testing peer as a response for a request,
- **Notification (2)** — a note sent by a testing peer as a reaction to an event.

The second part is the *Command* which identifies the action to be executed or, in some cases, that was executed. Currently, there are 18 command represented by a long integer. And the last part is the *Payload* which refers to the data carried by the note as part of its command. Detailed description of commands and its payload is available in Appendix A.1.

[4] Groovy — object-oriented programming language for Java platform http://groovy-lang.org/
[5] Messagepack — https://msgpack.org/

3.3 Measuring Process

After the dynamic test generation, with options from the test file, the measuring process starts. Senders will start sending messages to the SUT, while Inspector starts monitoring the behavior of the SUT and send measured data to the data server. For monitoring purpose, Inspector uses the Broker management interface — a REST interface that exposes (via HTTP protocol) an internal JVM[6] and Broker detailed information. The actual data collection by Inspector is straightforward:

1. Inspector sends a HTTP request with the *JavaScript Object Notation*[7] *(JSON)* content to the Broker REST interface.
2. Broker evaluates the request and sends response to the Inspector.
3. Inspector collects the response.

Note that errors occurred during the collection may cause the test case to fail.

However, there are two problem factors; the first is that the Inspector should not influence the performance of the SUT. Current solutions for the information collection works like the management interface method call with request for information and response retrieval. During this call, the method usually involves locks to guarantee the thread safety and exclusive access. However, calling this method too often can cause significant Broker performance degradation. In order to reduce this risk, the inspector enforces a collection interval of 10 seconds and restricts usage only to selected operations. This strategy reduces the hits on management interface to 2 or 3 hits every 10 seconds and presents suitable performance.

The other, problem factor, is the large size of the stored logs. This is mitigated by the usage of the compression methods. However, compressed logs can still fill the whole hard drive during the long test-run and so old logs has to be erased at some point of time. The collected logs can be safely erased when the test is completed. Currently the Maestro generates about 1 Gb of uncompressed data per hour of testing.

3.3.1 Testing Metrics

The type of metrics collected during tests depends on the cluster component. In the Table 3.1 we can see the summary of the metrics, which are collected for each component.

Table 3.1: The summary of Maestro metrics summary collected during test cases.

Component	Metrics	Description
Sender	Throughput	Throughput of the sender
Receiver	Throughput	Throughput of the receiver
	Latency	Time between send and receive messages
Broker	JVM heap memory	Maximum, minimum, and current Eden, Survivor, and Tenured space[8]
	JVM non-heap	PermGen or Metaspace
	Broker internals	Queue size and expiration count
	OS basic memory	Physical and swap memory usage
	OS resources	Count of file descriptors

[6] JVM — Java Virtual Machine
[7] JSON — https://www.json.org/

Throughput of the sender or receiver refers to the message count sent/received during the performance test run. This metric is collected by each sender and receiver. On the other hand latency is collected only by receiver. This refers to the time between sending and receiving of the message and can be influenced by the Quality of Service or other parameters. Since Messaging Broker is written in Java, JVM memory metric is relevant. High JVM memory usage can point to the memory leak or bad algorithm implementation. Broker queue has size threshold and message expiration time. When no one picks-up the message from the queue after some period of time there is no need to keep old messages and its unnecessary to fill too much of the memory.

Last metric is the OS resource spending during the performance testing. It is not relevant for broker performance, but it is helpful to know e.g. the CPU usage, memory usage, etc., in case of Broker crash debugging.

3.4 Collected Data Format

Data are collected by Inspector. Inspector continuously monitors the broker and collects information about the workload. Output of this measurement should be one file for each active inspector. The broker inspector file is composed of the following columns:

- **Timestamp** — the date and time of the data sample in the format YYYY-MM-DD hh:mm:ss using the W3C defined standard for datetime.
- **Load** — size of the system load.
- **Open file descriptors** — number of opened filed descriptors.
- **Free file descriptors** — number of free file descriptors.
- **Free memory** — free physical memory.
- **Free swap memory** — swap free memory.
- **Swap committed** — swap committed memory.
- **Eden initial** — Eden initial memory.
- **Eden committed** — Eden committed memory.
- **Eden max** — Eden maximum (limit) memory.
- **Eden used** — Eden used memory.
- **Survivor initial** — Survivor initial memory.
- **Survivor committed** — Survivor committed memory
- **Survivor max** — Survivor maximum (limit) memory.
- **Survivor used** — Survivor used memory.
- **Tenured initial** — Tenured initial memory.
- **Tenured committed** — Tenured committed memory.
- **Tenured max** — Tenured max memory.
- **Tenured used** — Tenured used memory.
- **PM initial** — Permgen or Metaspace initial memory (either Permgen or Metaspace depending the JVM version).

[8] Eden, Survivor and Tenured space are internal Java memory spaces.

- **PM committed** — Permgen or Metaspace committed memory (either Permgen or Metaspace depending the JVM version).
- **PM max** — Permgen or Metaspace maximum memory (either Permgen or Metaspace depending the JVM version).
- **PM used** — Permgen or Metaspace used memory (either Permgen or Metaspace depending the JVM version).
- **Queue size** — number of messages waiting for processing in the queue.
- **Consumers** — number of consumers connected to the queue.
- **Acknowledged** — number of acknowledged messages in the queue.
- **Expired** — number of expired messages in the queue.

Maestro sender and receiver generate another relative performance testing data. Receiver generates latency log with the following data:

- **Start Time-stamp** — start time of the receiving.
- **End Time-stamp** — end time of the receiving.
- **Interval Maximum** — collected maximum latency.
- **Interval Compressed Histogram** — compressed histogram of measurement's latency in HDR[9] format.

Both, sender and receiver generate rate (throughput) data files. These contain data about sent or received data by each peer. Data are stored in a compressed comma-separated values (CSV) file with the following columns:

- **eta** — represents the estimated time of departure/arrival of the message, relative to the start of the test.
- **ata** — represents the actual time of departure/arrival of the message, relative to the start of the test.

3.5 Related Works

One of the related tools for performance testing is *SpecJMS* [10]. It is the industry-standard benchamrk for evaluating the performance of eterprise message-oriented middlevare servers based on JMS.

[[optat se otavia proc je to naprd]]

[9] HDR — http://hdrhistogram.github.io/HdrHistogram/JavaDoc/org/HdrHistogram/package-summary.html

4 Analysis and Design

MPT is specially designed for the performance testing of Broker. However, with the Qpid-Dispatch growth, the need for performance testing comes. In the following we will analyze the Qpid-Dispatch service, focusing on its capabilities and methodology. Moreover we will describe the design of Topology Generator and Qpid-Dispatch Performance Module for MPT, which are the main requirements for actual performance testing of Qpid-Dispatch router.

4.1 Qpid-Dispatch Router

Qpid-Dispatch is a lightweight AMQP message router suitable for building scalable and highly performant messaging networks. This router is an application layer program, with respect to ISO/OSI[1] model, running either as a normal user program or as a daemon. In particular it has the following key features:

- Connects clients and brokers into an internet-scale messaging network with uniform addressing.
- Supports high-performance direct messaging.
- Uses redundant network paths to route around failures.
- Streamlines the management of large deployments.

The following summary of Qpid-Dispatch router was composed based on knowledge available in [22].

4.1.1 Theory of Operation

The router accepts AMQP connections from senders and receivers and further creates AMQP connections to Message Brokers or similar AMQP-based services. Through these connections sender is able to reach receiver, which can be another client in the network or a message broker. The client can exchange messages directly with another client without involving a broker at all. The router classifies all of these incoming messages and routes them between senders and receivers. The router is designed to be deployed in topologies of multiple routers, preferably with redundant paths to provide continually connectivity in the case any router in the network fails. For routing Qpid-Dispatch uses link-state routing protocols[2] and algorithms similar to OSPF or IS-IS to calculate the best path (e.g. the

[1] ISO/OSI — http://www.studytonight.com/computer-networks/complete-osi-model

[2] Link-state protocols — https://www.certificationkits.com/cisco-certification/ccna-articles/cisco-ccna-intro-to-routing-basics/cisco-ccna-link-state-routing-protocols/

path with the lowest cost) from sender to receiver through the whole network and to recover from failures.

4.1.2 Addresses and Connections

Qpid-Dispatch is able to connect client servers, AMQP services, and other router implementations through network connections. The router provides multiple components and settings for specifying the service on the other side of connection link as follows:

Addresses[3] — are used to control the flow of messages across a network of routers. Addresses can specify messages and they are also used during the creation of links since links are bounded to the specific address field of a source and a target. The address can refer to topics or queues that match multiple consumers to multiple producers. There are two types of addresses:

- **mobile** — the address is a rendezvous between senders and receivers. The router is message distributor.
- **link route** — the address is a private messaging path between sender and receiver. The router only passes messages between end points.

Listener — accepts client connections. Listeners have several types that are defined by their role:

- **normal** — the connection is used for AMQP clients using normal message delivery.
- **inter-router** — the connection is created to another router. Inter-router connection can only be established over inter-route listeners.
- **route-container** — the connection is established to a broker or other resource that holds known address.

Connector — is used as an interface for creation of connection with brokers or other AMQP entities using connectors. The same as listeners, connector has several types that are defined by their role:

- **normal** — the connection is used for AMQP clients using normal message delivery. The router will initiate the connection but links are created by the peer that accepts the connection.
- **Inter-router** and **route-container** — they are the same such as listener's modes.

To ensure the security the router uses the *SSL/TLS (Sockets Layer and Transport Layer Security)*[4] protocol and it is related certificates and *SASL (Simple Authentication and Security Layer)*[5] protocol mechanisms to encrypt and authenticate remote peers. Router listeners act as network servers and connectors act as network clients. Both of these components may be configured securely with SSL/TLS and SASL.

[3] Addresses in this discussion refer to AMQP protocol addresses, not to TCP/IP addresses.
[4] SSL — https://tools.ietf.org/html/rfc6101; TLS — https://tools.ietf.org/html/rfc5246
[5] SASL — https://tools.ietf.org/html/rfc4422

4.1.3 Message Routing

Addresses have semantics associated with them. These semantics control how routers behave when they see the address being used. There are two ways how the router can route messages based on addresses:

Routing pattern — define the paths which message with a mobile address can take. These routing patterns can be used in both case of message delivery; with broker or directly through the router.

- **Balanced** — anycast[6] method in which multiple receivers are allowed to use the same address.
- **Closest** — anycast method in which every message is sent along the shortest path to reach the destination.
- **Multicast** — method in which every receiver with the same address receives the copy of the original message.

Routing mechanism — define the path to endpoint from sender to receiver.

- **Message routed** — message delivery is done based on the address in message's *to* field. The router checks the destination address of the message and finds the same address in its routing table. The message is then sent to all links with that address.
- **Link routed** — this method uses the same routing table as Message routing with the difference that the routing occurs during the link-attach operation and link attaches are propagated along the appropriate path to the destination. This results into a chain of links from source to destination.

A message can be delivered with various degrees of reliability such as *at most once*, *at least once* or *exactly once*.

4.2 Automatic Topology Generator

For proper testing of the various messaging systems we need multiple topologies with different components and different settings. However creating and deploying the scenarios manually for each test scenario is rather slow and annoying, even after just few scenarios. The solution to this problem is divided into two parts: a simple topology generator, that transform metadata, defined by user, into configuration files for each component contained in metadata, and automatic *Ansible* scripts, which deploy the whole topology to actual physical machines. User has to define is a metadata file, a single file for the whole topology instead of single file for each component, and then start the Ansible script which ensures configuration files generation and the deployment.

4.2.1 Topology Components

Messaging system consists of multiple components with specific roles. In our case, testing topologies will consider clients, brokers, and routers. Clients refer to message senders and receivers, but there is no need for specific configuration for each client at all. Message settings is another case, but MPT deals with it as was mentioned at Chapter 3.

[6] Anycast vs. Multicast — anycast method sends data to every node in network, while multicast method sends data only to specified group of nodes.

Broker

Broker configuration file offers various settings and protocols such as specialized queuing behaviors, message persistence, or manageability. The following list shows selected capabilities of the broker:

- **User access** — allows other guest or authentication access to users.
- **Multiple Protocol Support** — broker supports AMQP, MQTT, STOMP, OpenWire and Core protocols.
- **Connections** — can establish connection to another AMQP-based service such as another broker or router.
- **Queues** — user can specify new queues in configuration file or allow auto-create option.
- **Messaging types** — refers to approach how to deliver messages, examples are point-to-point and publish-subscribe approach.
- **Logging level** — broker offers the setup for different logging levels.

However, broker configuration is not implemented yet, but the design of the automatic configuration generation will be shared with router configuration generation.

Router

Just like broker configuration, router offer various types of configurations. The basics were explained in Subsections 4.1.2 and 4.1.3, but for better understanding of all capabilities we recommend to refer to Qpid-Dispatch documentation [22].

4.2.2 Format of Input and Output

The format of the input should be user-friendly and easy to update even for large topologies. As the input we choose one file (`config.yml`) in *YAML*[7] language, which is similar to JSON format but it is better readable for humans. Topology Generator needs information about all hosts in the topology and which type of topology it should generate. For that purpose there are two attributes in the configuration file; first is the *inventory path* which refers to the location of *Inventory* — file, containing all hosts in topology in the specific format (for its specialization refer to Appendix B.1). It is a simple configuration file with enumeration of host names and their IP addresses. The second attribute is the type of the topology it should generate. The user can specify one of the simple types of graph, such as line, circle, complete, etc., which does not need any other information except Inventory or he can specify path to graph metadata, which are described in Subsection 4.2.3 in more details.

On the other hand, output format should be easy for automatic parsing. The best format for machine parsing in Ansible is JSON or YAML format, since both of them are loaded with same functions. Output of the generator will be then passed to an Ansible script immediately after the creation without any user intervention. However, user should have option to see the generator output in YAML format, because in case of larger topologies JSON is badly readable for human eyes. Hence output will be one JSON file with variables for template. Each node from Inventory will have its own variables separated from variables of other nodes. Scheme of the input and output for Topology Generator is shown in the Figure 4.1.

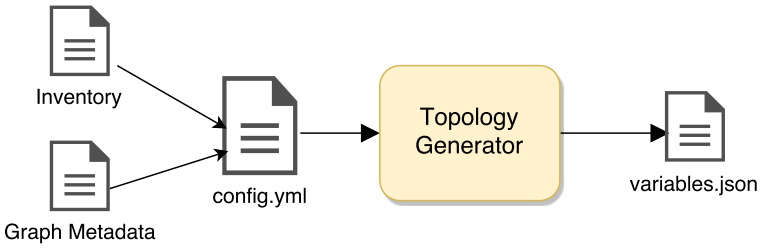


Figure 4.1: Input and output scheme of the Topology Generator.

4.2.3 Graph Metadata

The technology which will be used for the actual implementation of Topology Generator is *NetworkX*, a Python package for creation and manipulation of complex networks. This package offers features for creating graphs, multigraphs, random graph generators, plot created graph, and many more. NetworkX also offers graph import and export in YAML structured file, which is useful as graph metadata; simple example of this file is shown in Appendix B.2.

In this metadata user can specify any setting for each node. For example, user can specify the listener for router 1, or connector for router 2 as you can see in the example below.

```
---
directed: false
graph: {}
nodes:
- type: router
  id: router1
  listener:
   - host: 0.0.0.0
     port: 1080
     role: inter-router
- type: router
  id: router2
  connector:
   - name: router1
     host: router1
     port: 5675
     role: inter-router
multigraph: false
```

From this metadata NetworkX creates two nodes with type, id, and listener or connector attributes. These attributes will be used to generates configuration files for each node. All possible attributes that user can specify for each node are available in Appendix B.4.

However, specifying all attributes of each node is not very user-friendly approach, especially in case of large topologies. So user can only specify nodes, and links between them and generator will add all necessary attributes in order to establish connection between nodes. The example of this metadata file can be seen in Appendix B.2.

[7]YAML - http://docs.ansible.com/ansible/latest/YAMLSyntax.html

4.2.4 Topology Deployment

Every node specified in the Inventory has to receive proper configuration files for services running on it. This job is handled by the Ansible, since it can connect to all nodes from Inventory and copy configuration files to proper destination folders. Ansible script load data from Topology Generator and creates configuration files based on loaded variables and the common template for Qpid-Dispatch. The created file will then be sent to the proper node based on node name from Inventory, which has to be same like router name specified in generated variables. The scheme of configuration deployment is depicted in the Figure 4.2.

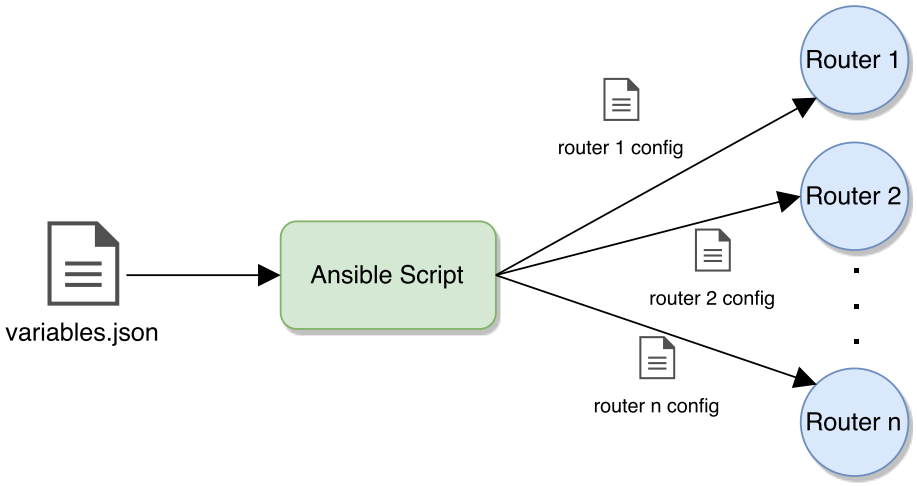


Figure 4.2: The scheme of configuration files deployment to the nodes.

4.3 Qpid-Dispatch Agent Performance Module

The architecture of Maestro (as depicted in the Figure 3.1) cannot use all performance testing and network recovery possibilities of the Qpid-Dispatch. For better performance analysis and measurements it is necessary to design and implement additional functionality for MPT.

In the Figure 4.3 we show updated version of Maestro architecture. Proper performance testing of router and network analysis with few routers needs some kind of an agent, which can manipulate each node. In particular, MPT should be able to shut down one of the router node and collect data about network behavior during this situation. All these actions will be handled by the new back-end component called *Agent*.

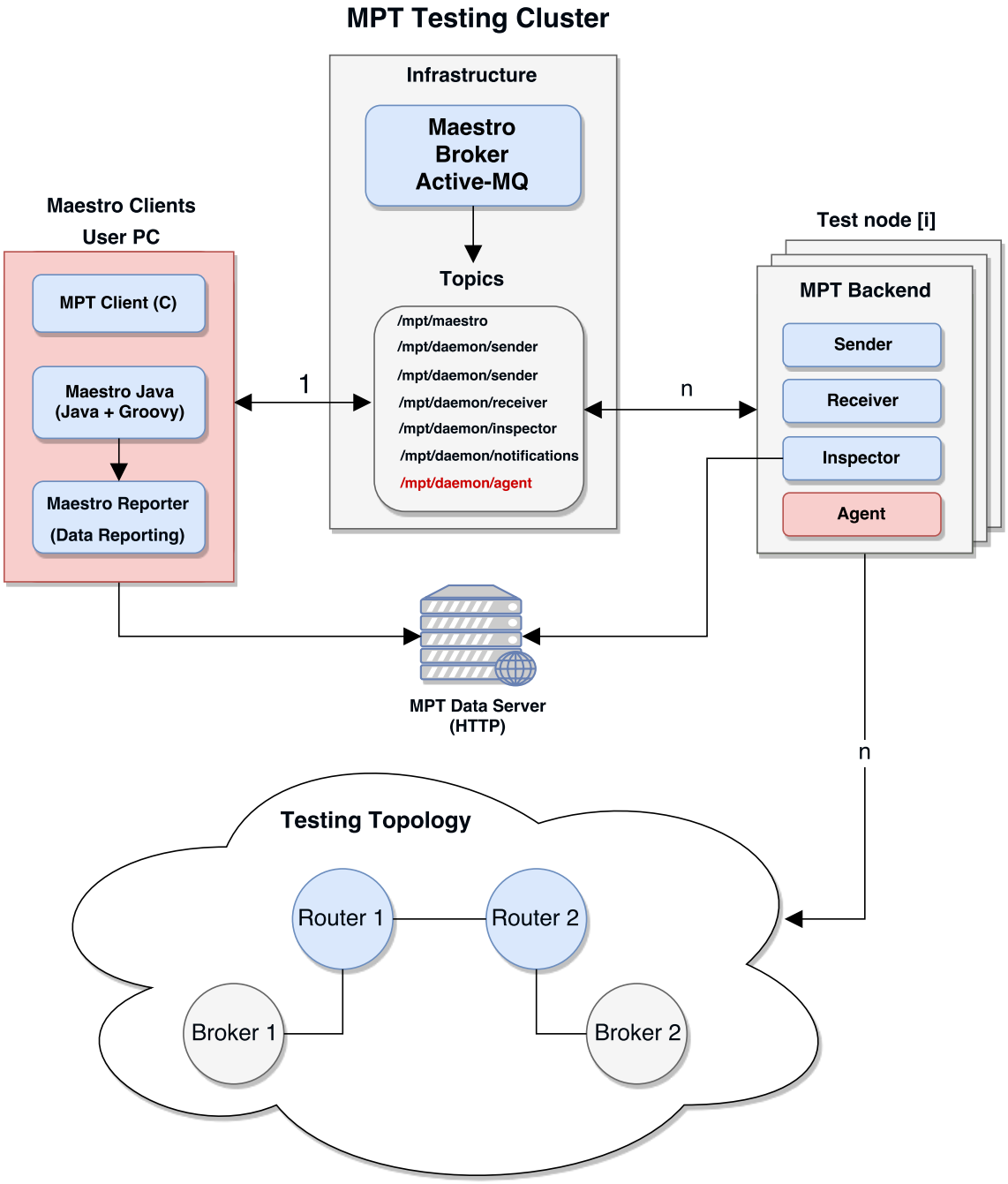
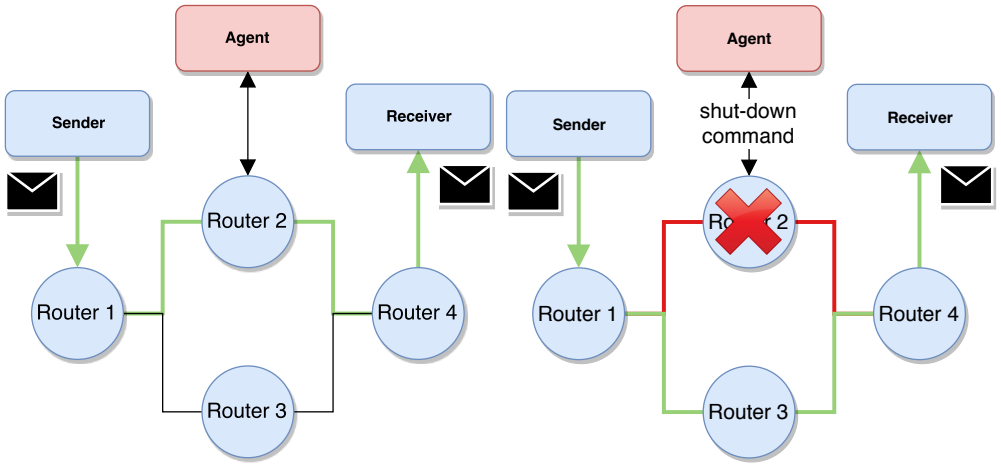


Figure 4.3: The architecture of updated Maestro for testing of the Qpid-dispatch router.

In the Figure 4.4a we show the simple scheme of topology and one agent monitoring the router 2. Communication passes through the router 2 and messages are delivered to receiver without problems. The Figure 4.4b demonstrates the shutdown of router 2 by the agent. In that case, the network will choose the redundant link through router 3 in order to pass messages. This scenario can answer the question *How does this incident influence the latency between sender and receiver?*



(a) Network with router agent. (b) Router shut-down demonstration.

Figure 4.4: Simple network with router shut-down demonstration.

Another part is a communication inside Maestro cluster with new component. Communication between cluster back-end and user client is realized through Maestro Broker and for proper message distribution new topic has to be added. As was mentioned in Section 3.4, Maestro Clients communicates with back-end via specialized commands. Router Agent will accept new set of commands for router control. This set has to be added to existing Maestro Clients. All additional components or components for update are highlighted by red color in the Figure 4.3. The example of simple testing topology consisting of two routers and two brokers is also included in the Figure 4.3.

4.3.1 Extension Points

After research and few discussions we decided to develop the agent as a service with dynamic command execution, which is able to run any specific code. At the begging of the test, the agent will receive the command for download some repository with specific scrips as a action handlers. The path to repository will be a payload of one of the new commands. After that, the agent will listen on the Maestro Broker topic for the agents and wait for user's command to execute. This command will transport the name of the handler script as a payload of the Maestro note. The agent will execute script from payload as an action on the node. In particular, the router restart handler can be part of downloaded repository and then can be performed after receive user commands with payload *r̈equests/router_restart.groovy¨*. This functionality makes the agent dynamic, which offers to user execute any specific action what he wants.

4.3.2 Communication with Agent

For the communication inside Maestro testing cluster the Maestro Protocol is used which was described in the Subsection 3.2. MPT Clients have to know how to communicate with this new component in the cluster and for this it is necessary to add new communication commands. The following list shows new commands which should be added:

- **MAESTRO_NOTE_START_AGENT (18)** — start the agent service.
- **MAESTRO_NOTE_STOP_AGENT (19)** — stop the agent service.

- **MAESTRO_NOTE_AGENT_SOURCE (21)** — path to user commands handlers.
- **MAESTRO_NOTE_USER_COMMAND (30)** — user's specific command.

4.3.3 AMQP Inspector

The important part of performance testing is measurements of internal metrics of SUT. Maestro offers Maestro inspector for this kind of measurements. However, current version can monitor only Broker, because Inspector is implemented for specific interface which is provided by Broker. Since Broker is written in Java and provides access to JMX[8] via Jolokia[9], we cannot use current Inspector for router as well. The router offers *AMQP management* for interaction with the router on the fly, which is different than Jolokia access. The Jolokia access is based on HTTP/JSON format message exchange between requester and SUT. AMQP Management is based on AMQP messages with specific format.

The router offers these information after proper AMQP request to an opened up listener with specific properties:

- **name** — this property is always setup to **self**.
- **operation** — AMQP management offers classic CRUD operations. For inspect message we will always use option called **QUERY**.
- **type** — this property represents interior package which will parse the request, we will use **org.amqp.management**.
- **entityType** — this property is configurable. We use there several options with prefix **org.apache.qpid.dispatch.** based on request purpose. The options are:
  - *router* — general informations about the router.
  - *router.stats* — detailed informations about the router.
  - *router.link* — informations about the route links.
  - *router.node* — general informations about neighbour nodes.
  - *router.address* — informations about addresses on the router.r
  - *connector* — informations about connections.
  - *allocator* — informations about memory metrics.
  - *config.autolink* — informations about created auto links.
  - *config.linkRoute* — informations about created link routes.
- **body** — this property is message payload, which is empty list. Exceptions are auto links and link routes requests, which needs additional information in the body.

Collected Data

Data collected by the AMQP Inspector is different than collect current version of Inspector. After the discussion, we decided to collect data about *general statistics*, *router links* and *memory*. Note, that each data set has multiple data columns, which are all available in Appendix C. The following describes the most important data collected by the AMQP Inspector:

[8] JMX — http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html
[9] Jolokia — https://jolokia.org/

- **Timestamp** — the date and time for the data sample in the format YYYY-MM-DD hh:mm:ss using the W3C defined standard for datetime.
- **General Statistics** — basic informations about the router such as active connections, addresses, auto links, accepted messages and so on.
  - **Address Count** — activate addresses at current time.
  - **Connections Count** — active connections at current time.
- **Router Links** — informations about all router links which were opened to the router.
  - **Accepted Message Count** — accepted messages at current time.
  - **Delivered Message Count** — delivered messages at current time.
  - **Released Message Count** — released messages at current time.
  - **Undelivered Message Count** — undelivered messages at current time.
- **Memory Statistics** — informations about allocated memory by the router.
  - **Total Allocated Memory** — total allocated memory.
  - **Memory Allocated by Threads** — total memory allocated by threads.

Each data set is converted to chart, which represents collected values for each request. Data collected by senders and receivers remains the same as in the current version of MPT.

4.4 Used Technologies

Messaging Performance Tool is a project with several parts. The most of MPT, such as command parsing, reporting, clients abstractions and so on, is written in Java language. But whole MPT is not pure Java code. To specify a test, Groovy is used. Groovy is basically lightweight version of Java with several advantages. From my point of view Groovy scripts are more readable for those who are not much familiar with Java code. Groovy scrips are also used as reaction for specific commands for extension points, but this is deeply described in the Subsection 5.2.1.

On the other hand, Topology Generator is a new simple project. For easy integration to another projects, quick development, and easy code preview I developed it in *Python* language. Whole generator is created as one package, which is available for installation on any machine with installed Python version 2.7 and higher. Already mentioned technologies are very common and almost every programmer have heard about Java and Python. In the following subsections I describe not common and well known technologies.

4.4.1 Ansible

Ansible [3] is simple automation framework which allow users to automate daily tasks on multiple nodes or containers. Basic types of tasks which can be automated by Ansible are:

- **Provisioning** — setups the various servers user needs in the network infrastructure.
- **Configuration management** — changes configuration of an application, operation system or device. Basically this allows starting, stopping and restarting services, installing or updating applications or performing a wide variety of other configuration tasks.

- **Application deployment** — automatically deploys the internally developed application to specified systems with all dependencies.

Ansible scripts, called playbooks, are written in YAML language. This makes Ansible scripts easy to read for humans and simple to manage. Another advantage is that the user does not even need to know commands used to accomplish a particular tasks. All that is needed is to specify what state does user wants the system to be in. Ansible is available on multiple systems with really short list of dependencies; Linux based systems requires Python installed and Windows needs PowerShell; both systems requires SSH. Moreover, Ansible playbooks can be grouped together and create more complex scripts called roles. They are open-source and available in public repository.

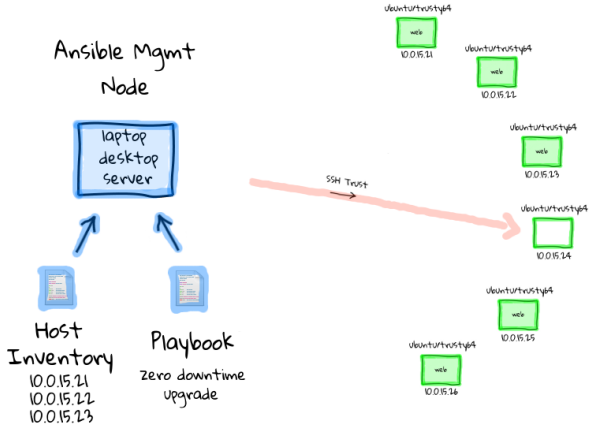


Figure 4.5: Ansible architecture. [[Placeholder - dobre popsat]]

We use Ansible used for several tasks; mainly to deploy systems on specific nodes. As I want to run performance tests of Qpid-dispatch over multiple topology scenarios it is necessary to do system deployment quickly and automatically, which is easy with Ansible. System deployment contains installation of MPT, Qpid-dispatch and other services based on testing scenario. The next task is to create and deploy configuration files for each router machine. This task runs Topology generator and creates configuration files for each machine based on generator output.

4.4.2 Docker

Docker [1] is an open platform that provides developing, shipping, and running application separately from the infrastructure. Basically Docker is a specific type of virtualization technology. It allows to package and run an application in a loosely isolated environment called a container. These containers are lightweight virtual machines running directly within the host machine's kernel. This means that one can run more containers than virtual machines on specific hardware, and it is possible to run containers on virtual machines.

Docker container is build up from a dockerfile where container attributes are specified such as OS, environment variables, or steps for installing applications. Output of build command is then a docker image. This image is ready for running as a container with another specific attributes such as exposed ports. Containers can be attached to same network which allow communication between all containers.

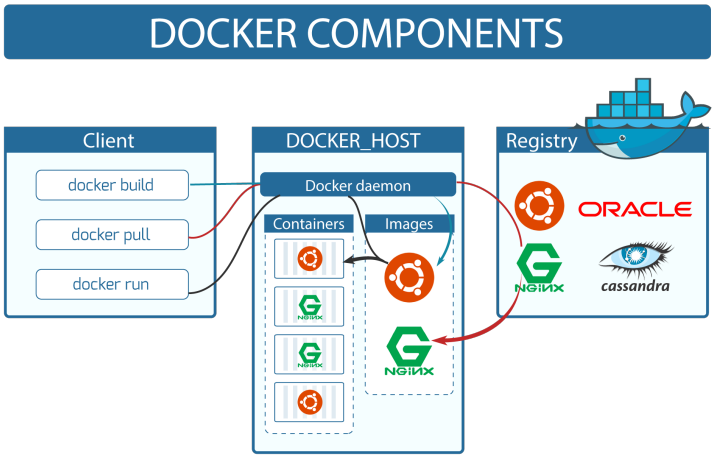


Figure 4.6: Docker architecture. **[[Placeholder]]**

Since docker is able to run services such as Qpid-dispatch very easily and also allow communication between containers. It is possible to deploy MPT with proper SUT in containers and analyze behavior in the container network or just run MPT on single machine. However, for proper performance results we need real machines, docker containers are used only for MPT development and trying some basic stuffs with MPT.

5 Implementation

This chapter describes the actual implementation details of components, described in the Chapter 4. The main part focuses on the Agent module and AMQP Inspector for MPT, which is implemented in Java and Groovy language. The other part describes the Topology Generator, — python package for automatic generation of dispatched topology based on user's metadata. Measurement data gathering and reporting done by MPT parts has already been mentioned in the Chapter 4.

5.1 Topology Generation

Qpid-dispatch has a lot of configurable attributes, which can influence the router behavior. These attributes can be set up with an AMQP management tool called *qdmanage*[1] or one can specify them directly in the configuration file. However, qdmanage needs human interaction, it is more comfortable to create configuration file for each specific test case. Hence, this initiates implementing automatic Topology Generator.

In case of network with multiple routers, it is uncomfortable to update configuration files for each router on a specific node. Topology Generator introduces an option to update only a single file with router specifications and leave generation and deployment to an automated script. The actual generation takes few simple steps to achieve correct configuration files. These steps are used in Ansible script and are described in the following.

5.1.1 Configuration File Generation

It is important to note that each configuration file is not generated by Topology Generator itself, but by Ansible playbook. Why do we need this approach? Since Qpid-dispatch is getting new versions every few months, they can change names of any configuration attribute or even erase them. This causes the problem, that when Qpid-dispatch is updated, then the code of Topology Generator has to be reviewed and updated too, otherwise one risks syntax errors in the configuration files. Such approach is not very stable, and hence the simple solution is to let Ansible do the final generation.

The trick is, that Ansible is able to fill-up any kind of passed *Jinja2*[2] template only with data which are available. Basically, the Ansible playbook will get the configuration template and variables for router configuration files and creates proper configuration file. The script simply iterates through template and fills-up all available attributes. This process is repeated for every router machine in the Inventory file. Configuration variables are in JSON format, Ansible can recognize which variables are for particular machine.

[1] qdmanage — https://qpid.apache.org/releases/qpid-dispatch-1.0.1/man/qdmanage.html

[2] Jinja2 — modern and designer-friendly templating language for Python http://jinja.pocoo.org/docs/2.10/

5.1.2 Template Generator

Output configuration files are strictly based on input configuration template. This means that Ansible needs input template with specific attributes for each version. However, Qpid-dispatch offers a solution how to construct this template. Attributes are available inside a JSON file in the installation folder of Qpid-dispatch. To process this JSON file and create resulting configuration template we use a tool called *qdrouter-jinja2*[3].

Qpid-dispatch configuration file is divided into the multiple section where each sections has its own attributes. For example there is a *router* section with router name, mode, etc., and *ssl* section with security attributes. Each section can be specified multiple times, but usually only the last one found is used. The exceptions are *connectors*, *listeners*, *addresses* and *link routes* that can specify multiple connection points and routing types on single router. In the Algorithm 1 you can see pseudo-code of template generation process.

Input: *attributes_file* — input file in JSON format
Output: output file in Jinja2 format

```
1  var output = ""
2  for line in attributes_file :
3  |   if line.is_attribute() :
4  |   |   output += line.attributeToJinja2()
5  |   else if line.is_section() :
6  |   |   output += line.sectionToJinja2()
7  |   else
8  |   |   output += line
9  output.strip()
10 return output
```

Algorithm 1: Template generation by qdrouter-jinja2.

From the pseudo-code you can see that there are two kind of wrappers for processing the JSON. Their function is to make configuration sections and attributes optional and repeatable which is achieved by wrapping the sections and attributes with Jinja2 code. The attribute wrappers processes each attribute line into the following template snippet:

```
{%% if section.attribute is defined %%}
    attribute: {{ section.attribute }}
{%% endif %%}
```

This code in template specifies, that if Ansible knows the variable *section.attribute*, it will add a line with that attribute name and variable value into the configuration file. Key words section and attribute are just placeholders for real names such as *connector* for section and *host* for attribute. Output can then look like the following line:

```
host: 10.0.0.1
```

The section wrapper is more complex, because it has to wrap the start and the end of the section. This is handled by class methods `_enter_()` and `_exit_()` which allows you to implement objects that execute `_enter_()` at start and `_exit_()` at the end of some statement. Basically this class is created for each section and these methods are invoked before first and after last attribute. The `_enter_()` method wraps start of each section with following code:

[3] qdrouter-jinja2 — https://github.com/rh-messaging-qe/qdrouter-jinja2

```
{%% if item.section_name is defined %%}
{%% for section_name in item.section_name %%}
section_name {
```

The **__exit__()** method closes the section with the following piece of code in the Jinja2 template:

```
}
{% endfor %}
{% endif %}
```

Since qdrouter-jinja2 parses JSON data from installed version of Qpid-dispatch on remote node it guarantees that the template will always correspond with specific router version. The template is saved in */tmp* folder on the remote machine where Ansible scripts can fetch it into the local folder and fill it up with data.

5.1.3 Topology Generator

Topology Generator is the main part of configuration generation and deployment. It process configuration variables for Ansible deployment scripts from the user specification. Topology Generator requires two parameters: the path to the Inventory and path to the graph file or topology type:

Path to the Inventory — Inventory is simple configuration file with list of nodes, connected to the network. Generator retrieves node names and types (router, broker) and use them during generation of variables. The generator creates specific sections and attributes based on node and graph types. Since broker configurations are not generated by this tool, it uses information only about specification of link routes to neighbours. Broker configuration is based on XML files, where user can specify Broker attributes. However, the future goal is to generate configuration for Broker too.

Path to Graph file — Graph file is a simple YAML file which specifies node distribution in the network. It contains at least node name and links to another nodes. Beside the name, user can specify another node informations such as constructors, listeners, SSL profiles, etc. easily for each node. The whole file is loaded during initialization and is processed with the Topology Generator.

Topology Type — Topology generator can create topology without graph file, but then it requires network type which will be generated. For example the topology type can be a line which puts all nodes into one line and generates connections between them.

Inner representation of network is realized by Python library *NetworkX*[4]. It creates a graph as an object and offers manipulation with its attributes which are objects of nodes and links. Topology Generator is able to store information about network configurations as attributes of these objects. During the graph initialization, the generator stores basic information about nodes such as the name and the type from inventory or some additional information from the graph file. Basic algorithm of topology generation is depicted in the Algorithm 2.

[4]NetworkX — https://networkx.github.io/documentation/latest/

However, the generation for each configuration section is more complex and is slightly different for sections for connections to another nodes. The generation is split into two parts based on the user's arguments: the first is generation of the default connections and the second is the generation of user specific sections from metadata file.

Default Connections — default connections corresponds to configuration for establishing connection between two devices in the network. To achieve this one has to configure listeners, connections, addresses and link routers (depending on second machine) on each router. This sections can be easily automatically generated only with the minimal knowledge about the network. The default connections are generated automatically when user specifies only hosts and topology type. The generator takes neighbours of each machine. Generator's output in that case is a file with variables for fully functional connections between machines. During the generation from the graph file each node has attribute which specifies if user wants the default connections. The Algorithm 2 captures the default generation process.

User Specific Sections — these sections are not needed for the proper communication inside the network. An example of this section can be SSL or auto-links settings. The generator loads data about these sections from graph file. Qpid-dispatch has a lot of setting, hence the generator does only the basic connectivity configuration without any specific settings if the user does not specify otherwise. You can see the user specific sections generation in the Algorithm 2 as the part of the first *for* statement. This generation part is done alongside with default connections generation.

Used algorithms are pretty straightforward. Since the generator is able to load IP addresses from inventory there has to be mechanism for automatic generation of proper port numbers for listeners and connectors. The problem is, that connectors of node X and listeners of directly connected node Y has to have same port numbers. It means, that node X connects to a specific port on node Y and node Y listens on that port. The initial port numbers is 5672, the default AMQP port, and it is incremented with each newly created listener. Hence, the listeners must be generated first on all nodes and then the connectors can be generated. This allows the access to port numbers of neighbor listeners via simple method and explains the double loop over nodes in the Algorithm 2.

Input: Inventory, Graph File/Topology Type
Output: output file in JSON format

```
1  var inventory = parse_inventory(Inventory)
2  var graph = create_graph(inventory, Graph File/Topology type)
3  var output = {}
4  for node, neighbors in graph.adjacency() :
5  |   output.update(generate_listeners(node, neighbors))
6  |   output.update(generate_addresses(node, neighbors))
7  |   output.update(generate_specific(node, neighbors))
8  for node, neighbors in graph.adjacency() :
9  |   connectors, link_routes = generate_connectors(node, neighbors)
10 |   output.update(connectors)
11 |   output.update(link_routes)
12 return output
```

Algorithm 2: Default connectivity generation.

Function: *generate_connectors()*
Input: *node*—node from graph, *neighbors*
Output: lists of connectors and link_routes

```
1 var connectors = []
2 var link_routes = []
3 for neighbor in neighbors :
4 |   if neighbor.is_router() :
5 |   |   connectors.append(connector_setting)
6 |   else if neighbor.is_broker() :
7 |   |   connectors.append(connector_setting)
8 |   |   link_routes.append(link_route_setting)
9 return connectors, link_routes
```

Algorithm 3: Connectors and link routes generation. The algorithm describes function `generate_connectors()`.

The Algorithm 3 shows the generation process of connectors and link routers. The connectors are generated for each other network service (router/broker), but link routes are generated only in the case of the connection to the broker. The link route section then contains name or address of the connected broker, name of queue to which router will send the messages and specification of link route direction (input or output). For full-duplex connection to the broker one needs connector and two link routes from the router to the broker.

5.1.4 Deployment

At this point, everything is ready for creation the Ansible playbook to run all necessary tools and do the deployment of generated configuration files. Note, that each task can be executed on different machine based on the inventory.

The playbook combines all previously mentioned tools and also uses some features from Ansible role *ansible-qpid-dispatch*[5] such as start/stop handlers. These steps are added in any playbook or role, and can be used for automatic topology generation and deployment. The necessary things are Inventory and topology metadata for each test-case. In the following description you can see the list of deployment steps, that are executed on the control node (node where use the playbook):

1. **Install the Topology Generator** — Topology Generator is the main actor in the topology deployment so it is necessary to have it installed. Ansible takes care of it in the playbook.

2. **Run the Topology Generator** — Topology Generator needs configuration files for proper execution. In play one just needs to specify the path to configuration files and Ansible will do all other things.

3. **Include variables into Ansible** — this step loads the generated variables into memory. After this step, the script is ready to fill-up the template on remote machines.

Since Ansible offers smart system with variables inside the playbooks, one can assign all paths to configurations files to variable in the script or pass them with option during

[5] Ansible-qpid-dispatch — Ansible role for install and setup Qpid-dispatch. The role is available at https://github.com/rh-messaging-qe/ansible-qpid-dispatch

the playbook execution start. After these steps we are ready to execute few steps on the remote nodes:

4. **Install qdrouter-jinja2** and **generate templates** — this qdrouter-jinja2 is used to generate template. We need to install it on all of router nodes, because each router can have different version and it can affect the configuration file with deprecated attributes. After installation the templates are created.

5. **Fill templates on remotes** — the script fills-up the template on each node. Since it has information about all nodes from configuration variables, it simply compares hostname with key from variables to assign proper data to each host.

6. **Restart Qpid-dispatch** — after the change of configuration, we need to restart each machine and reload the configuration.

5.2 Qpid-Dispatch Performance Module

This section focuses on Maestro Agent implementation and updates of all other Messaging Performance Tool parts such as commands updates, extension of the Inspector and report changes. The Agent is implemented in Java and Groovy languages.

5.2.1 MPT Preparations

The first step during the development is to update the Maestro project structure by adding the new module called *maestro-agent*. The agent is designed as the new independent service, which can be run after the building of the package by Maven. First, we need the main function for the agent, whoch is built with each new package. After the creation of main we have to create *assembly.xml* which tells Maven which files has to be used for creation of new package during the build. The last step is to update all *pom.xml* files, where are specified all dependencies and then, we are ready to build and start the implementation.

5.2.2 Agent Module

As it was mentioned in Subsection 4.3.1, the agent is an independent service running on the testing node. Since Maestro already has a similar services, we can reuse the already working parts. The Maestro has a class `MaestroWorkerManager` which represents a simple Maestro peer. This class has a several important methods which are inherited and used by Agent as well:

- `connect()` — this method connects each peer to the Maestro Broker. Based on the peer function, it also subscribes the peer to all needed topics. For example, the sender peer does not need subscription to agent commands topic. When this method throws an exception, the peer is not able to connect to Maestro Broker and the test fails.
- `noteArrived()` — this method catches incoming notes from Maestro Broker and invokes action based on the note.
- `handle()` — this method handles each received note. We overload this method to invoke specific handle method based on the received note type. Usually, the `handle()` methods in `MaestroWorkerManager` class only logs actions. For another functionality they are overridden in specific implementations of each peer.

Every action handle script is written in groovy, and so we needed a groovy script executioner. For this purpose, we created the class `GroovyHandler`. This class basically checks the handler file if it is executable and then tries to execute it. The handler file location is specified by the note payload. In that location one can specify more than one file; `GroovyHandler` checks and execute all of the files.

The main part of the Agent is the method called `callbacksWrapper()`. Since the Agent overrides `handle()` method to execute scripts from external point, every `handle()` method in the agent calls the `callbacksWrapper()`. The basic functionality is shown in the Algorithm 4. The reason why `sendReplyOk()` is sent everytime is that we need to know if thread was started. For example we can start the thread with the command execution 5 minutes after the start. We need the information if thread started successfully and then the information how the thread execution finished. However, the information about thread finish is sent by the handler itself. This is also reason why every external point handler creates its own thread. Naturally, the agent must serve other handlers during this time, and not wait 5 minutes for the finish of one of them and then handle the others.

Input: externalPointPath, codeDir, note
Output: sendReplyOk() or sendReplyFail()

```
1 var thread = Thread()
2 thread.start(groovyHandler.runExternalPoint())
3 try
4 |   var groovyHandler = GroovyHandler()
5 |   groovyHandler.setInitialPath(externalPointPath)
6 |   groovyHandler.setMaestroNote(note);
7 catch
8 |   sendReplyFail()
9 sendReplyOk()
```

Algorithm 4: Basic functionality of `callbacksWrapper()` method. [[**Urcite vylepsit**]]

The other important method of Agent is the `handle()` override for *AgentSourceRequest* note. After this note is received, the `handle()` method fetches a git repository URL from the note and tries to clone it. The current version offers to clone any public git repository and even the specific branch of the repository.

Agent Capabilities

The current implemented version of the Agent offers much more functions than was originally designed. The Agent does not focus only on Qpid-dispatch actions handling, but it can invoke action on node itself by executing extension points scripts. This makes agent usable also for Broker nodes, where it can simulate a real network behavior during the testing. The agent can also run 3rd party software on the node during the test, which can simulate any kind of the unexpected behavior.

The agent is specific kind of Maestro Worker. This means, that agent connected to the Maestro Broker can publish messages during the test about its execution status or any additional information. You can see a simple communication with agent notes handling in the Figure 5.1. The notes are send from front-end through Maestro Broker. Agent then invokes a specific handle method based on the received note. Inspector keeps inspecting the Qpid-dispatch by requests about his state every 15 seconds.

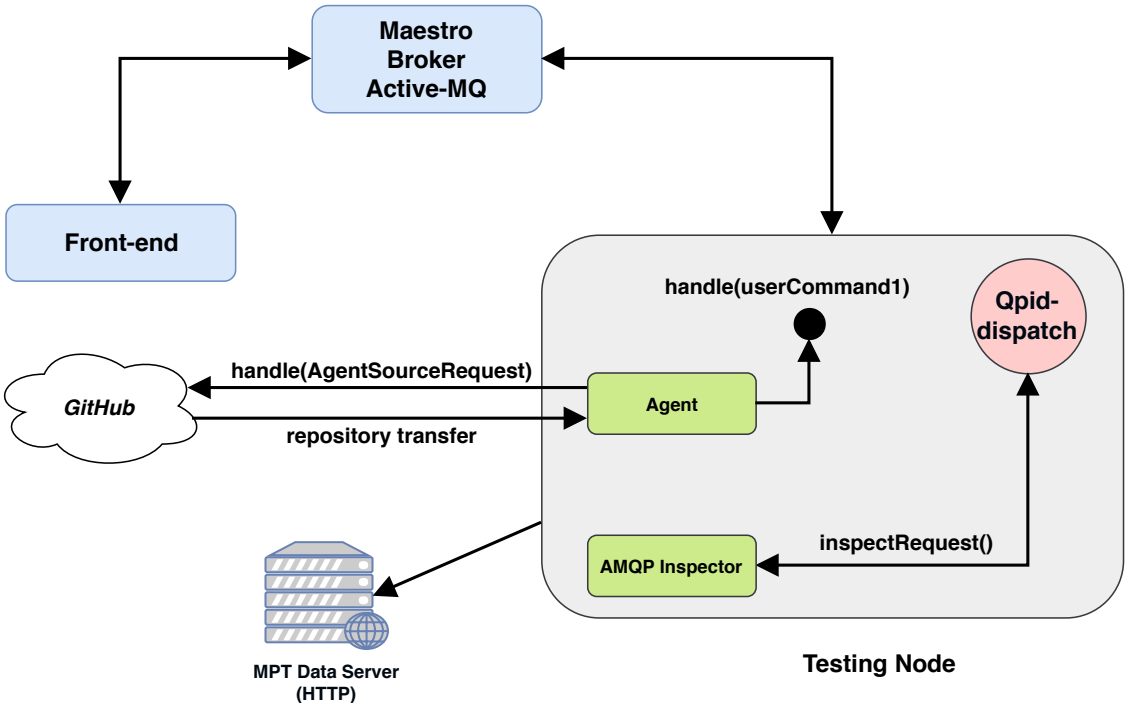


Figure 5.1: Communication inside the Maestro with agent notes handling.

[[dopsat neco?]]

5.2.3 AMQP Management Inspector

The collection of information about the router itself are not gathered by the agent. For this purpose, we developed a new type of Maestro Inspector specific for AMQP Management. AMQP Management is layered on top of the AMQP protocol and it access the inner data about the router by a simple requests and responses. Qdmanage tool has implemented operations for AMQP Management, however qdmanage is a Python tool and we want to integrate only Java code with AMQP Management requests into the Maestro. AMQP Management offers CRUD operations for router configuration and inter informations. For AMQP Inspector we are fine with only Read operation to get specific information about running the instance of Qpid-dispatch.

Basic Evaluation

The Maestro Inspector is designed to run specific Inspector class based on users definition in testing script. Currently, Maestro offers ActiveMQ Inspector for the Broker and AMQP Inspector for the Router. The Inspector will receive note with *inspector start* command, which carry string payload. This payload is name of specific inspector implementation that will be started. The start inspector mechanism is depicted in the Figure 5.2 and in the Algorithms 5 and 6.

Figure 5.2: Inner mechanism of Maestro Inspector during the receive start inspector note.

Function: *handle()*
Input: Maestro note — startInspector
Output: sendReplyOk() or sendReplyFail()

```
1 var inspectorClass = note.getPayload()
2 try
3 |   var inspector = Inspector(inspectorClass)
4 |   var thread = Thread(inspector)
5 |   thread.start() sendReplyOk()
6 catch
7 |   sendReplyFail()
```

Algorithm 5: Handler method for startInspector note which creates instance of specific inspector implementation.

Function: *start()*
Output: sendReplyOk() or sendReplyFail()

```
1  var routerLinkInforWriter = RouteLinkInfoWriter()
2  var memoryInfoWriter = MemoryInfoWriter()
3  var generalInfoWriter = GeneralInfoWriter()
4  try
5      var currentTime = System.currentTimeMillis()
6      connectToRouter()
7      var dataReader = DataReader()
8      while canContinue() do
9          routerLinkInforWriter.write(currentTime, dataReader.collectRouterInfo())
10         memoryInfoWriter.write(currentTime, dataReader.collectMemoryInfo())
11         generalInfoWriter.write(currentTime, dataReader.collectGeneralInfo())
12         Thread.sleep(5000)
13     end
14     sendReplyOk()
15 catch
16     sendReplyFail()
```

Algorithm 6: Method for start new implementation of the Inspector. This method continuously send requests to the SUT, collects, parse and write the response into csv file.

The AMQP Inspector uses of request-response message mechanism with the SUT. The inspector creates message with Java library *Qpid JMS*[6]. The message specification was described in the Subsection 4.3.3. Since we want to collect as much relevant data as possible, we are sending four request-response messages with different entityType option every 5 seconds during the whole test. For response collecting it is necessary to create temporary queue. That queue is used by the router as and response destination. The destination is contained in field *response-to*. The request message is represented as an object with type *JMS Message*. The whole request-response process mechanism is depicted in the Figure 5.3 and in the Algorithm 6.

[6] Qpid-JMS — https://qpid.apache.org/components/jms/index.html

Figure 5.3: Request-response mechanism of AMQP Inspector and the SUT.

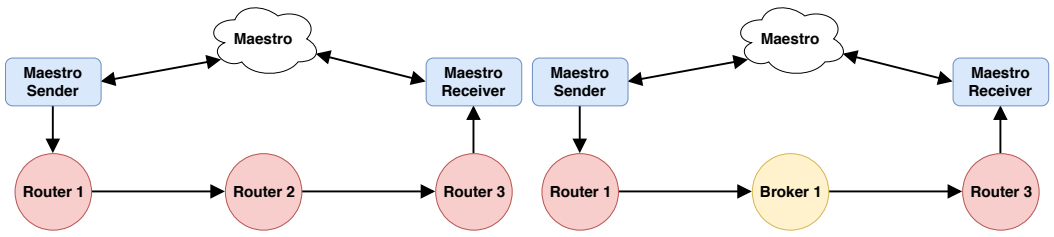
6 Experimental Evaluation

This chapter summarizes collected results during the performance testing and experiments with Messaging Performance Tool. We decided to split the testing into two parts. The first part is basic measurement with Maestro 1.2.4. During this measurements we measured the highest possible throughput of singlepoint topology of Qpid-dispatch and multipoint topology with three nodes of Qpid-dispatch or with Broker in the middle. The topologies are depicted in the image 6.1. The later series of experiments were focused on behavior testing of the topologies. For this measurements, we used Maestro version 1.3.0 which includes Maestro Agent and AMQP Inspector.

Since the testing was done over multiple topology types, we used Topology Generator for quick automatic changes of topology. All measurements was orchestrated by an automation server called Jenkins[1].

6.1 Basic Performance Measurements

Maestro works as the orchestration system, and requires proper infrastructure before we could run any test for our experimental evaluation. The architecture of Maestro, described in the Chapter 3, specifies that in ideal scenario one needs at least four machines for running a simple test: maestro broker, sender, receiver, and SUT. The amount of needed machines rises with more complex scenarios and larger networks. Examples of generated topologies are depicted in the Figure 6.1. For both of these configurations we compared the throughput and latency of these combinations and discuss the results with supervisors and author of MPT.



(a) Topology consists of routers. (b) Topology with Broker in the middle.

Figure 6.1: Topologies created for basic performance testing and experiments.

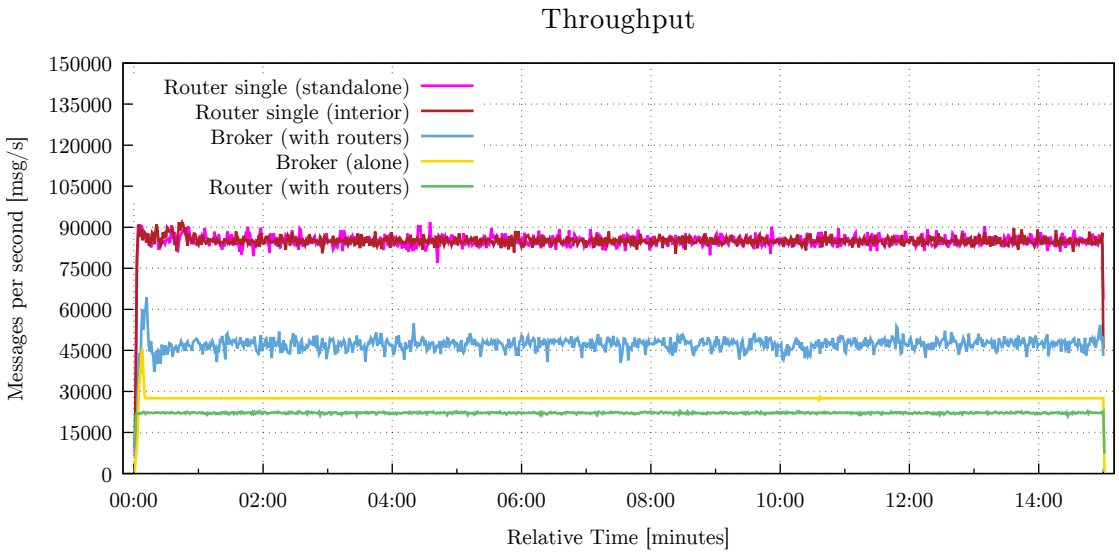


Figure 6.2: Chart of the maximum throughput of router and broker during specific test cases.

6.1.1 Throughput

We measured throughput only by load generators — *Maestro Sender* and *Maestro Receiver*. Load generation depends on the test properties. Maestro is able to create an unbounded rate test, during which it generates as much load as it can. This type of test was used to reach the maximum handled rate of Qpid-dispatch and its result is depicted in the Figure 6.2. However, the maximum rate was not achievable on all topology types. Qpid-dispatch offers two modes — *standalone* and *interior*, where the standalone works as a single router machine in the network, while interior type works with multiple routers. The current version of Qpid-dispatch has an ability to load balance when buffers are almost full. This ability offers faster message delivery in cost of a slower throughput. We compared standalone and interior router as depicted in the Figure 6.2 as the first case study. One can see that **[[dopsat popis obrazku]]**

But, throughput can also be influenced by other network devices. As you can see in the Figure 6.2, the lone router (pink and red color) can reach around 90 000 messages per second, while lone broker reaches only about 30 000 messages per second. However, when we try to add another component, the throughput changes significantly. The topology with three routers uses the flow-control of interior setting which cause perceptible throughput degradation of that network (shown by green color). On the other hand, when we replace router in the middle by broker, the throughput is raised to 50 000 messages per second. Thus one can see that load balancer of Qpid-dispatch should be improved, because throughput degradation is too high as we demonstrated. However, It has to be said that high throughput does not necessarily mean better performance. We discuss this in the Subsection 6.1.2.

[1] Jenkins — https://jenkins.io/

6.1.2 Latency

Latency is measured only by Maestro Receiver from certain load samples. Since the Broker is a distribution service, which needs to store messages for some time, or create and keep queues for clients, it has higher requirements for system resources. On the other hand Qpid-dispatch has only one purpose — to route the messages. This makes it more faster than the Broker. So high load can be unprofitable, especially in the case of topology with broker. The broker can handle less messages than router, but using router can raise broker's throughput since it can control the load. Thus it gives more time to broker to process messages even with higher load.

In the Figure 6.3 you can see the latency difference that we measured between those two services. You can see the measured latency on specified topology of three routers (red), and two routers with some middle-broker (blue). The latency curve proves, that router is able to deliver messages into its destination faster than broker, because broker needs to store them in the memory. The latency of the topology with broker reaches more than 1 000 μs; on the other hand, topology consisting of routers has significantly better latency that tops around 256 μs. Note, that both of those tests were run on the same machines and with the same test setting: 10 000 000 messages, 80% of maximum rate with five parallel senders and receivers and 256 byte message size.

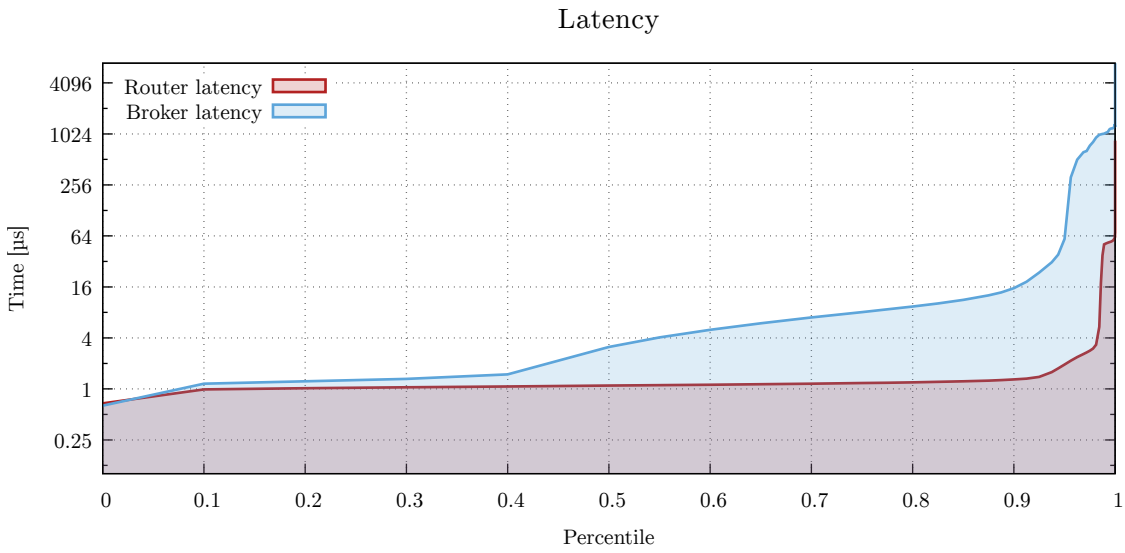


Figure 6.3: Latency chart showing the difference between router and broker latency at 80 % of maximum rate. The router's latency is significantly better than latency of Broker.

Another interesting comparison is between the latency of single instance router and broker. In the Figure 6.4 one can see the latency output from the test with the same load of 70 000 messages per second. Broker can reach this rate with additional queue settings. This configuration improves the distribution of incoming load between multiple queues. That causes that Broker is slightly faster than the router in 40% cases, but even with that, router is faster in other cases. Router latency has threshold around 60 μs, while Broker's has threshold over 1 000 μs. The conclusion is, that router should be much faster than Broker during certain circumstances.

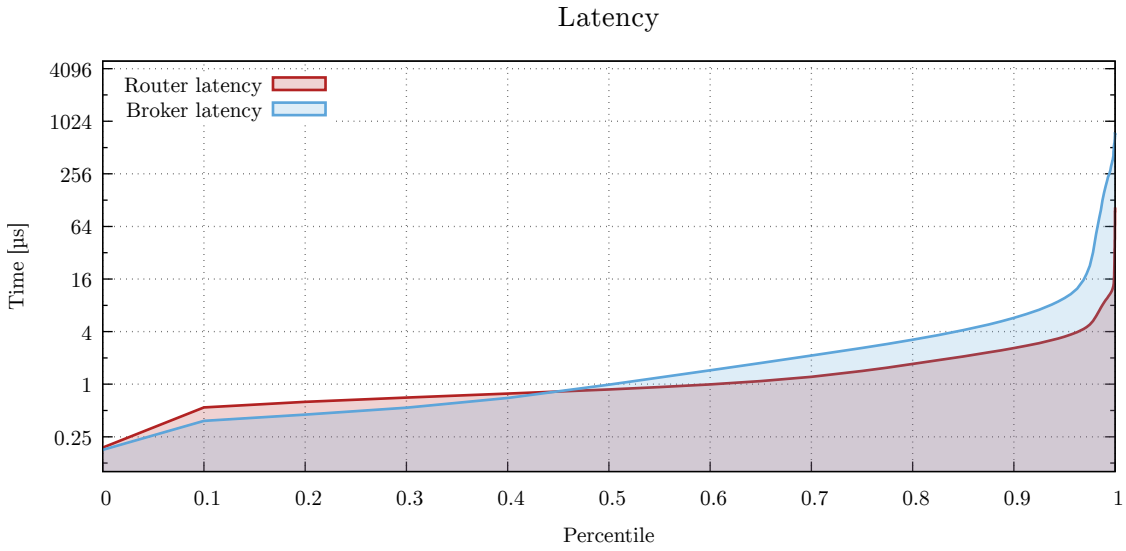


Figure 6.4: Router and broker latency comparison during the same load. You can see that router's latency is more stable than latency of Broker.

6.2 Behavior Measurements

[[Tohle popsat az s nejakym dalsim merenim, zatim asi netreba kontrlovat, pouze prevzato z Excelu]]

6.2.1 Agent Evaluation

Moreover, we present some preliminary results with using the agent extension and changing the behavior of topology depicted in the Figure 6.1. In the Figure 6.5 you can see throughput of few simple tests during which middle router is restarted or shut down. The throughput spikes are caused by these events. Since router does not have any queues to store messages, the messages are then discarded and lost. However, the sender does not receive acknowledgment of lost messages so it is not router responsibility. In the Table 6.1 you can see the duration of each executed operation and rate of lost messages during the operation (with expected amount of messages being 10 000 000). For example during the restart, router was completely shutdown for a second during which no messages arrived. However, after the restart there was some time to balance the load to the previous point. This leads to message lost equals to 2 seconds rate.

Table 6.1: Summarization of lost messages during the connection issues.

Operation	Duration (seconds)	Message Lost
None	0	None
Restart	1	46437
Shutdown	12	280572
Long Shutdown	89	1304451

The Figure 6.5 also shows, that test case with long shutdown is longer about 10 seconds that other scenarios and the throughput after the shutdown is quite higher. This means that router high throughput to even the messages lost.

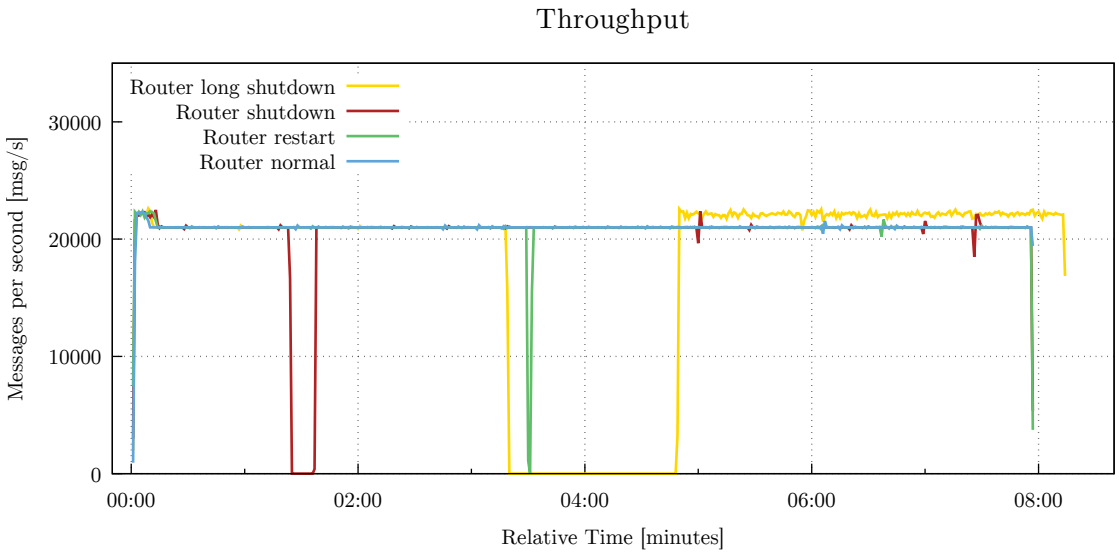


Figure 6.5: Router throughput comparison during the same load after different unexpected events.

Latency of test cases cases with the agent function demonstration is depicted in the Figure 6.6. You can see that router is able to even the latency during the restart and short shutdown with test run without any unexpected behavior. On the other hand, long shutdown (red) gets worse latency almost for 50 percentile of messages.

6.3 Performance Testing on Various Generated Topology

6.4 Testing results

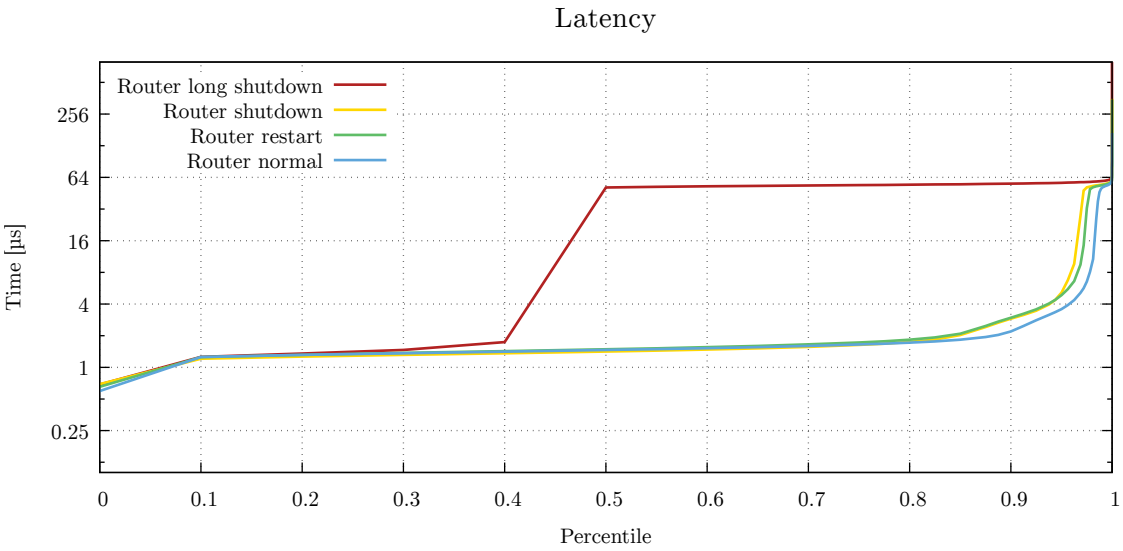


Figure 6.6: Router and broker latency comparison during the same load.

7 Future work and ideas

8 Summary

In this report we described the fundamentals of performance testing, common performance metrics and bugs and selected related tools. Further, we introduced the architecture and functionality of Messaging Performance Tool (MPT), and proposed its extension necessary to performance testing of Qpid-Dispatch tool. Moreover, we designed and implemented the Topology Generator tool, which is going to be used for semi-automatic topology configuration generation, which will significantly simplify the testing phase. The source code of the topology generator can be found in Appendix B.5.

In the follow-up work, the core of the thesis — a module for Qpid-Dispatch performance testing — will be implemented, integrated into MPT and the architecture and functionality of Topology Generator will be refined to allow precise performance testing of Qpid-Dispatch tool together with fine data analysis. The work will be concluded with experimental evaluation based on implemented modules and tools.

The results of this measurements was described and published in the paper for Excel@FIT[1] conference.

[1] Excel@FIT — IT conference for students and theirs work http://excel.fit.vutbr.cz/

Bibliography

[1] Docker. Online. [visited 2018/03/11].
Retrieved from: https://docs.docker.com/engine/docker-overview/

[2] ISTQB Foundation Level and Agile Tester Certification guide. Online. [visited 2017/11/29].
Retrieved from: http://istqbexamcertification.com/

[3] Network Automation with Ansible. Online. [visited 2018/03/11].
Retrieved from: https://www.ansible.com/overview/networking

[4] Regression Testing. Online. [visited 2017/11/15].
Retrieved from:
http://softwaretestingfundamentals.com/regression-testing/

[5] Software Testing Dictionary. Online. [visited 2017/11/15].
Retrieved from: https://www.tutorialspoint.com/software_testing_dictionary

[6] Anukool Lakhina, C. D., Mark Crovella: Diagnosing Network-Wide Traffic Anomalies. Online. [visited 2017/11/13].
Retrieved from: http://www.cs.bu.edu/fac/crovella/paper-archive/sigc04-network-wide-anomalies.pdf

[7] Bhatt, N.: Performance Testing – Response vs. Latency vs. Throughput vs. Load vs. Scalability vs. Stress vs. Robustness. Online. [visited 2017/11/5].
Retrieved from: https://nirajrules.wordpress.com/2009/09/17/measuring-performance-response-vs-latency-vs-throughput-vs-load-vs-scalability-vs-stress-vs-robustness/

[8] Broadwell, P. M.: Response Time as a Performability Metric for Online Services. Online. [visited 2017/11/19].
Retrieved from: http://roc.cs.berkeley.edu/papers/csd-04-1324.pdf

[9] Buch, D.: 4 types of load testing and when each should be used. Online. [visited 2017/11/5].
Retrieved from: https://www.radview.com/blog/4-types-of-load-testing-and-when-each-should-be-used

[10] Corporation, S. P. E.: SpecJMS. Online. [visited 2018/01/03].
Retrieved from: https://www.spec.org/jms2007/

[11] Curry, E.: Message-Oriented Middleware. Online. [visited 2017/12/21].
Retrieved from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.418.173&rep=rep1&type=pdf

[12] Din, G.: *A Performance Test Design Method and its Implementation Patterns for Multi-Services Systems*. PhD. Thesis. Technical University of Berlin. Berlin, Germany. 2008.

[13] Gao, J.; Ravi, C. S.; Raquel, E.: Measuring Component-Based Systems Using a Systematic Approach and Environment. Online. [visited 2017/10/26]. Retrieved from: https://subs.emis.de/LNI/Proceedings/Proceedings58/GI.Proceedings.58-6.pdf

[14] Kopp, M.: Why Averages Suck and Percentiles are Great. Online. [visited 2017/11/20]. Retrieved from: https://www.dynatrace.com/blog/why-averages-suck-and-percentiles-are-great/

[15] Manzor, S.: Application Performance Testing Basics. Online. [visited 2017/10/26]. Retrieved from: http://www.agileload.com/docs/default-document-library/application-performance-testing-basics-agileload.pdf

[16] Marko Aho, C. V.: Computer System Performance Analysis and Benchmarking. Online. [visited 2017/11/15]. Retrieved from: http://www.cs.inf.ethz.ch/37-235/studentprojects/vinckier_aho.pdf

[17] Martina, K.: *Unified Reporting for Performance Testing*. Master's Thesis. Brno University of Technology, Faculty of Information Technology. Brno. 2017.

[18] Molyneaux, I.: *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc.. first edition. 2009. ISBN 0596520662, 9780596520663.

[19] OASIS: *Advanced Message Quieing Protocol (AMQP) Version 1.0*. 2012.

[20] Piske, O. R.: Messaging Performance Tool. [Online; visited 2017/10/15]. Retrieved from: http://orpiske.github.io/msg-perf-tool

[21] Red Hat, Inc.. Raleigh, North Carolina, U.S.: *Red Hat JBoss AMQ 7.0 Using AMQ Broker*. 2017. available at https://access.redhat.com/documentation/en-us/red_hat_jboss_amq/7.0/pdf/using_amq_broker/Red_Hat_JBoss_AMQ-7.0-Using_AMQ_Broker-en-US.pdf.

[22] Red Hat, Inc.. Raleigh, North Carolina, U.S.: *Red Hat JBoss AMQ 7.0 Using AMQ Interconnect*. 2017. available at https://access.redhat.com/documentation/en-us/red_hat_jboss_amq/7.0/pdf/using_amq_interconnect/Red_Hat_JBoss_AMQ-7.0-Using_AMQ_Interconnect-en-US.pdf.

[23] Sharma, D.: Why and How: Performance Test. Online. [visited 2017/10/26]. Retrieved from: http://www.qaiconferences.org/tempQAAC/Why%20&%20How-Performance%20Test.pdf

A The Maestro Protocol

[[Dopsat podle zaverecne verze commandu]]

A.1 The Maestro Commands

Complete commands description is available on the following link as a part of Maestro documentation.

https://github.com/orpiske/msg-perf-tool/tree/master/doc/maestro/protocol

B Topology Generator

B.1 Inventory

An example of Inventory file for Topology Generator and Ansible deployment scripts.

```
[clients]
sender ansible_host=10.0.0.1
receiver ansible_host=10.0.0.2

[routers]
router1 ansible_host=10.0.0.3
router2 ansible_host=10.0.0.4

[brokers]
broker1 ansible_host=10.0.0.5

[nodes:children]
brokers
clients
routers
```

B.2 Graph Metadata

Example of graph metadata file for Topology Generator. Generator will generate graph with 2 routers and 3 brokers where routers are connected together and each broker is connected to one router.

```
---
directed: false
graph: {}
nodes:
- type: router %node type
  id: router1 %node name
- type: router
  id: router2
- type: broker
  id: broker1
- type: broker
  id: broker2
```

```
links:
- source: router2 %source node for link
  target: router1 %target node for link
- source: router2
  target: broker2
- source: router1
  target: broker1
multigraph: false
```

B.3 Topology Generator Output

Example of Topology Generator output in YAML format. This output is for two directly connected routers.

```
---
confs:
- machine: router1
  router:
  - id: router1
    mode: standalone
  listener:
  - host: 0.0.0.0
    role: inter-router
    port: 6000
  - host: 0.0.0.0
    authenticatePeer: 'no'
    role: normal
    port: 5000
    saslMechanisms: ANONYMOUS
  connector:
  - host: router2
    role: inter-router
    port: 6001
  address:
  - prefix: closest
    distribution: closest
  - prefix: multicast
    distribution: multicast
  - prefix: unicast
    distribution: closest
- machine: router2
  router:
  - id: router2
    mode: standalone
  listener:
  - host: 0.0.0.0
    role: inter-router
    port: 6001
```

```
- host: 0.0.0.0
  authenticatePeer: 'no'
  role: normal
  port: 5001
  saslMechanisms: ANONYMOUS
connector:
- host: router1
  role: inter-router
  port: 6000
address:
- prefix: closest
  distribution: closest
- prefix: multicast
  distribution: multicast
- prefix: unicast
  distribution: closest
```

B.4 Qpid-Dispatch Configuration File Template

Template for configuration files for current version of Qpid-Dispatch is available at https://github.com/rh-messaging-qe/ansible-qpid-dispatch/blob/master/test/files/templates/qdrouterd-roland.conf.j2.

B.5 Topology Generator Source Code

Complete source code of Topology Generator is available at https://github.com/rh-messaging-qe/iqa-topology-generator.

C AMQP Inspector Data Sets