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
AUTOR PRÁCE
Bc. JAKUB STEJSKAL

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. TOMÁŠ FIEDOR,

BRNO 2017

Abstract

[[Doplnit]]

Abstrakt

[[Doplnit]]

Keywords

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

Klíčová slova

Testování, analýza 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 bachelor's thesis was prepared as an original author's work under the supervision of Mr. X The supplementary information was provided by Mr. Y All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references. [[**dopsat**]]

. . . .
Jakub Stejskal
January 6, 2018

Acknowledgements

Here it is possible to express thanks to the supervisor and to the people which provided professional help (external submitter, consultant, etc.).

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) [15] 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* [15] 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) [17]. MPT is specialized for the performance testing of *Message Broker* (Broker) [18] — network application level software cooperating with *Qpid-dispatch service* [19] 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 [15, 14, 9, 1].

2.1 Performance Testing Process

The main goal of the performance testing is to ensure the following application attributes [10]:

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.),

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.

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 of performance tests and execution 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.

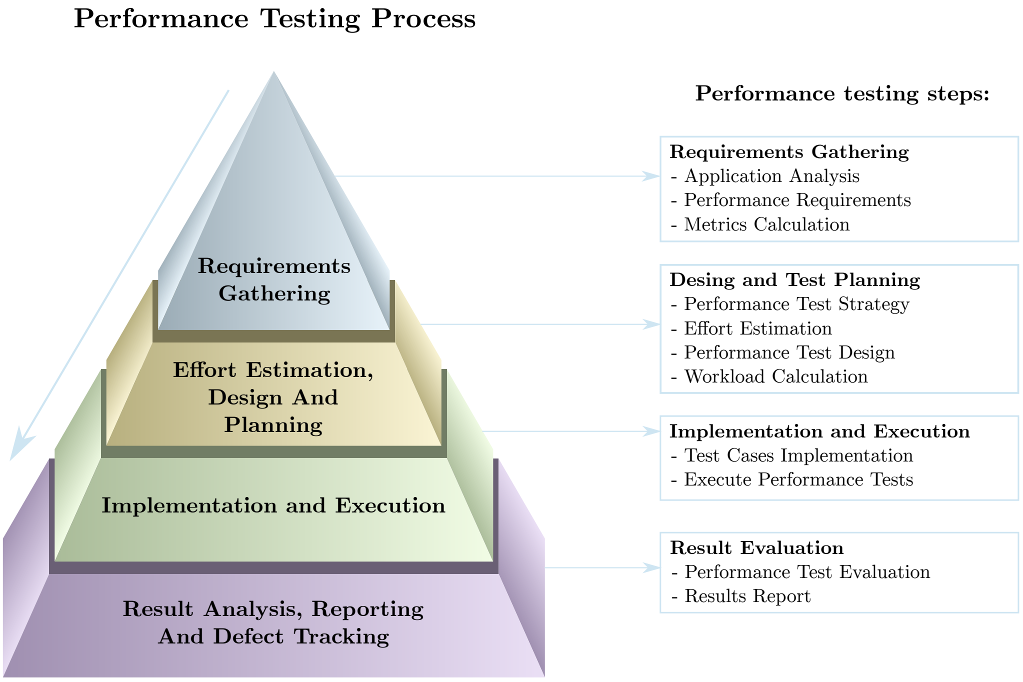
Performance Testing Process
Requirements Gathering
Effort Estimation, Design And Planning
Implementation and Execution
Result Analysis, Reporting And Defect Tracking
Performance testing steps:
Requirements Gathering
- Application Analysis
- Performance Requirements
- Metrics Calculation
Desing and Test Planning
- Performance Test Strategy
- Effort Estimation
- Performance Test Design
- Workload Calculation
Implementation and Execution
- Test Cases Implementation
- Execute Performance Tests
Result Evaluation
- Performance Test Evaluation
- Results Report

Figure 2.1: Performance Testing Process with the four most important parts and theirs individual steps based on [20].

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 - S.U.T*, suitable performance metrics, that will model the application performance, and set performance requirements. 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 S.U.T 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 hardware machines for testing and installing the necessary software for running load injectors, tested components, 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 and their fix may take a long time.

While this process seems trivial, the opposite is true, in particular 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 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, we 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* [7, 12] 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, hence its necessary to run this type of tests during network 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, memory consumption or machines handles,
- 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 using components.

Response Time

Response time is time 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* [12] 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.

However response time limits for messaging system are more strict. They could acquire values in milliseconds or less.

[[Prvni iterace]]

Traffic Spikes

As *traffic spike* [14, 4] we can understand the sudden degradation of one of the performance metrics such as *throughput, bandwidth, error rate, response time* but also resource usage such as *memory usage, disk space, etc.* during the network traffic. 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.

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 method for testing of spikes is called *stress testing* [12] and it is described in Section 2.3 in more details. Network system should also be scalable, 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.

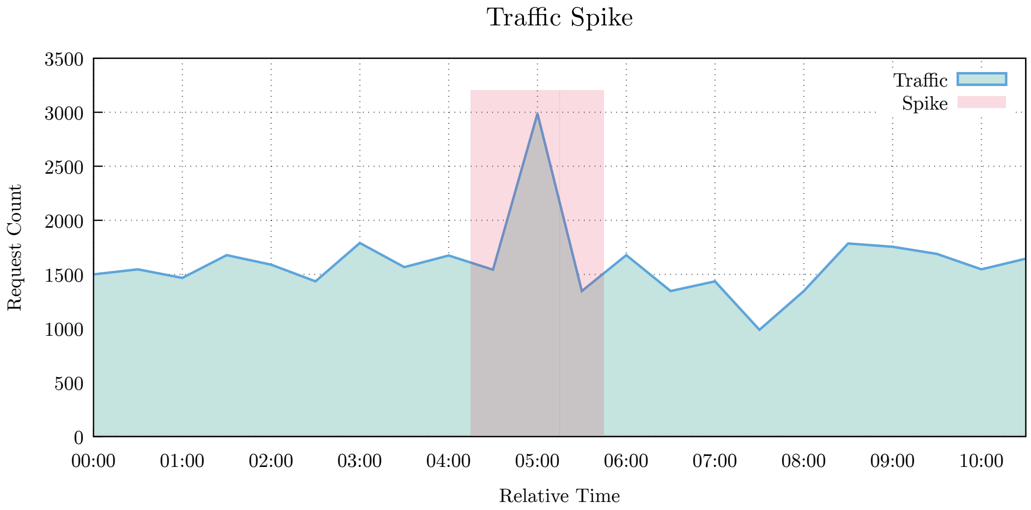

Figure 2.2: The graph shows amount of concurrent sessions depending on time. During to network traffic monitoring the traffic spike occurring after 5 hours from test start.

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. Their description is based on knowledge available in [3, 7, 15, 1].

Load Testing

Load testing 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 S.U.T. 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 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.

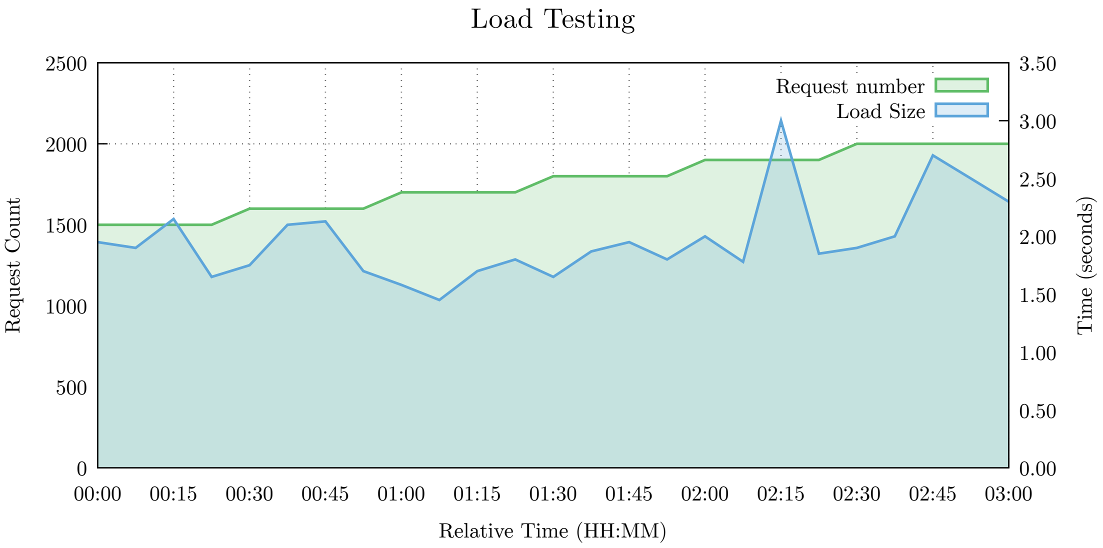

Figure 2.3: Response time of the system during the load testing depended on by load size.

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.

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 maximum number 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:

- **[[todo]]**

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. With higher load there comes performance issues which leads to unexpected rise of the response time.

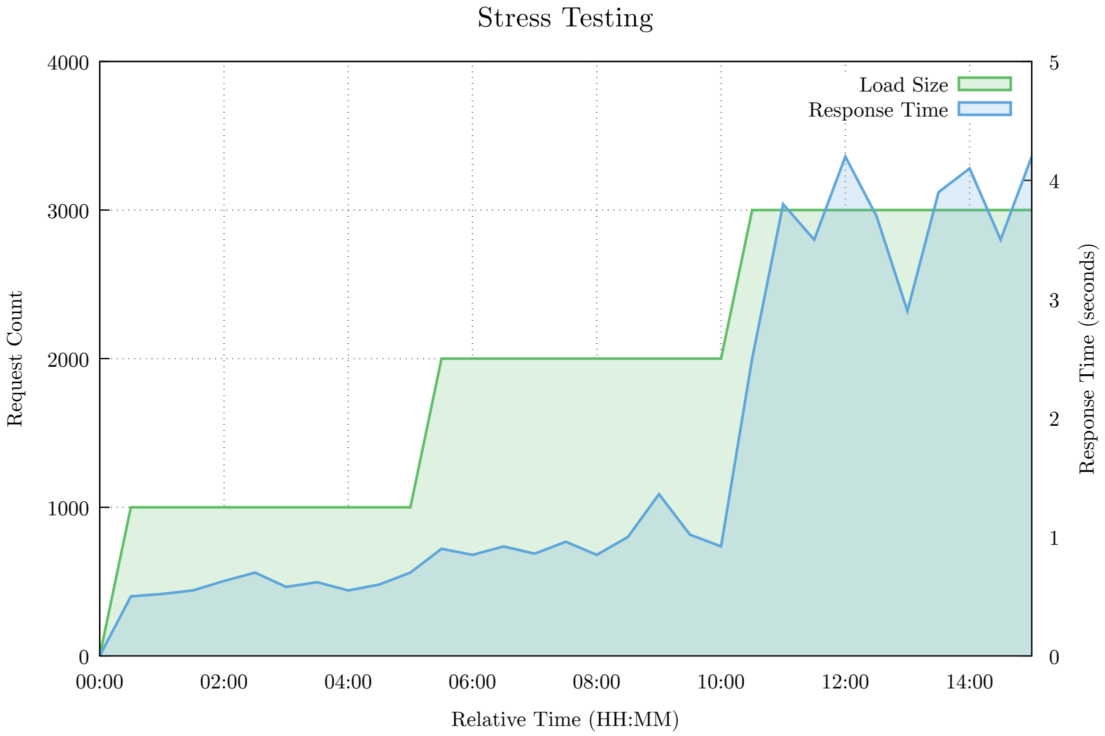

Figure 2.4: Stress testing diagram capturing dependency of response time on mount of requests.

Soak Testing

Soak, or stability/endurance 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.

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.

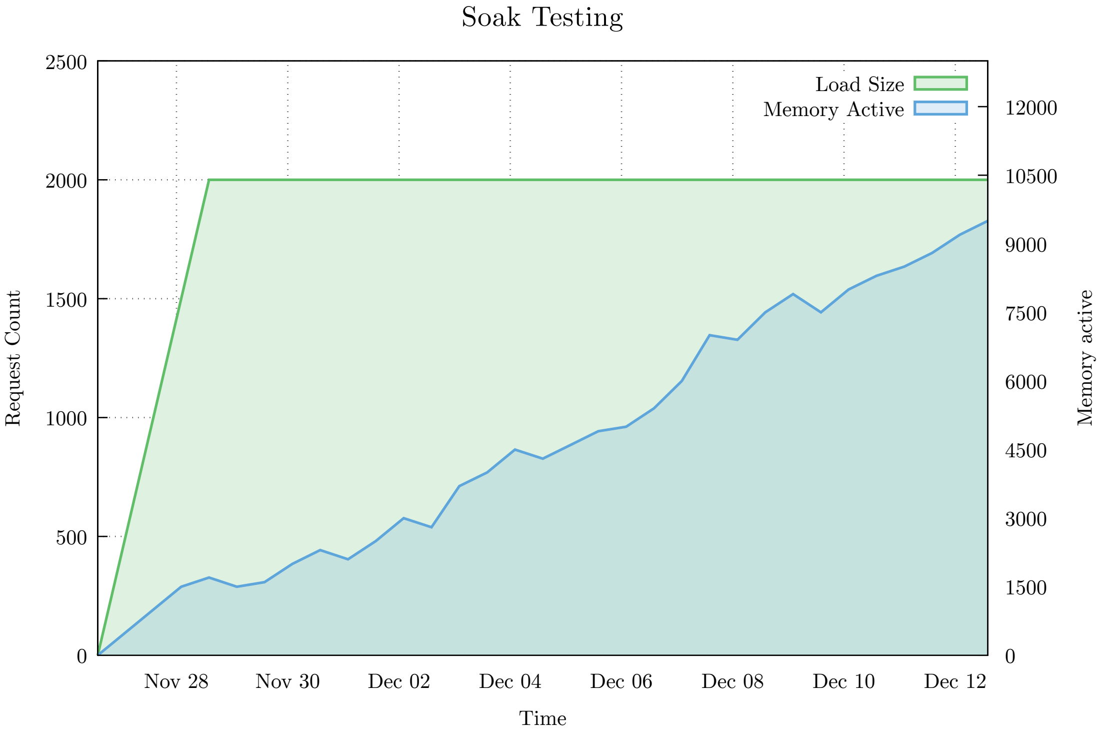

Figure 2.5: Soak testing with memory usage dependent on time.

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

Smoke Testing

Smoke testing method 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 release.
- **[[Anything else?]]**

Typical system issues covered by smoke testing testing:

- System without main functionality is useless.
- Main functionality can result into system crash.

Regression Testing

Whenever engineers develop new feature and want to update the previous build it has to pass the *regression tests* [2]. 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 [13] is method, which collects performance data during the system run on different hardware machines. Collected data 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.

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 network 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.

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 usually measured after a warm-up period of time after the commencement of traffic, because it takes a while for workload to stabilize. We need stabilized workload for proper throughput measurement, because initial values could negatively affect the measurement. Throughput is measured by load testing; suitable strategy for measuring throughput is to continuously raise the load until response takes longer that acceptable threshold.

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 ti 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

Second part of the response time is latency [6, 5], 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.

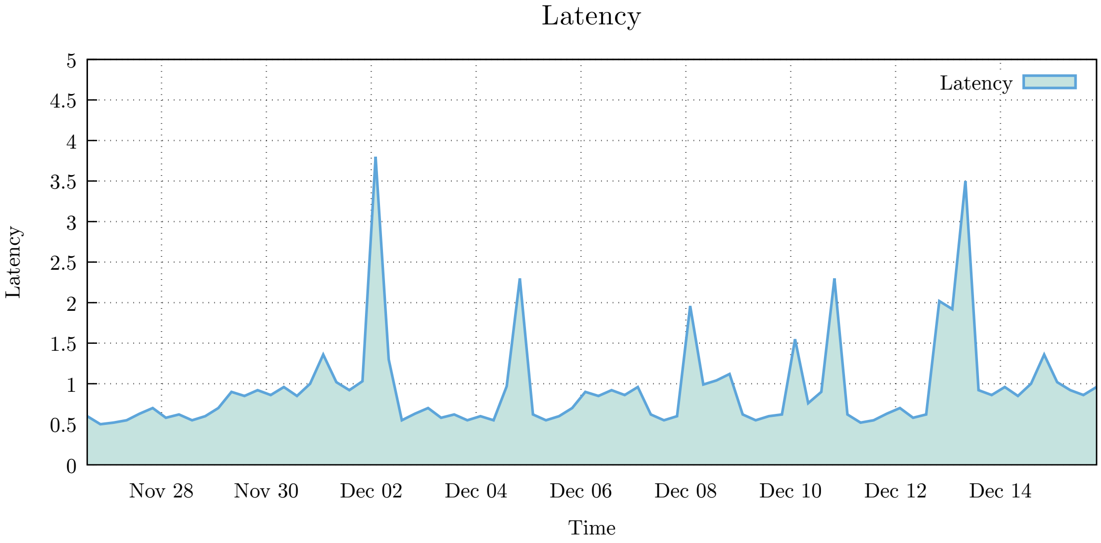

Figure 2.6: Latency diagram with system's response time based on the date.

Average and Percentile Response Time

There are two common ways of measuring the response time [11]: 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 outlines such as very slow transactions. In the Figure 2.7 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.

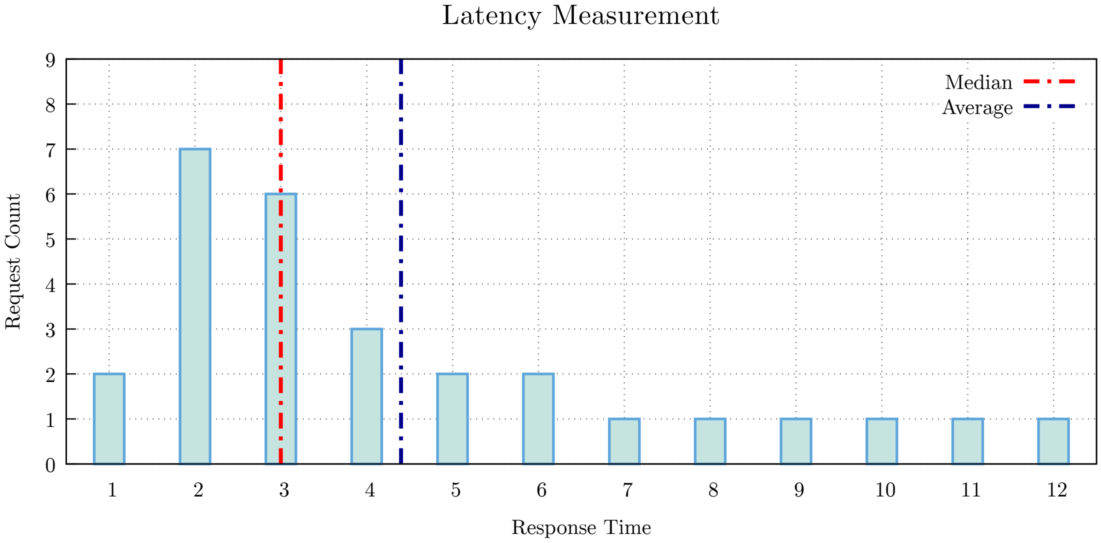

Figure 2.7: Transactions response time with calculated average and median of response time. Average represent inaccurate response time in this case which is higher than real one.

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 2.7 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.

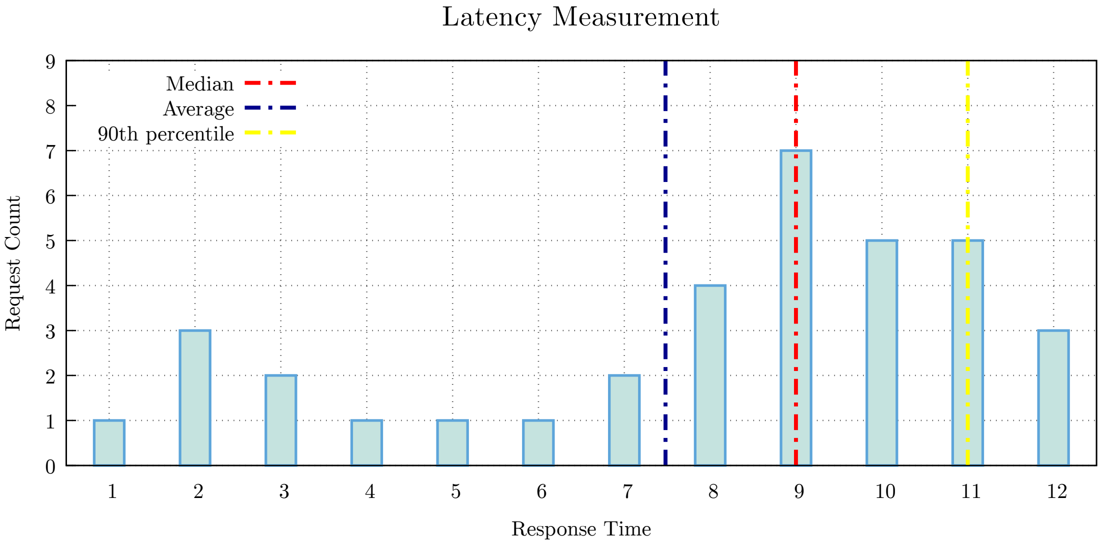

Figure 2.8: Transactions response time with calculated average and median of response time. Average represent inaccurate response time. Average says, that STU is faster than is in reality.

The Figure 2.8 shows different situation. 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 this case, 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.

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, hub, etc.) or signal disturb which can corrupt 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) [8] is one of the most critical elements of quality assurence for enterprise integration system. There are multiple messaging components developed in Red Hat such as clients, Message Broker, Message Router (Qpid-dispatch service) and stream-like message distributions tools. **[[Kafka - free to disclose?]]**

A Message Broker is an example of MOM. Its purpose is receive, store and distribute messages, which are sent and received by clients. The performance capabilities of a Message Broker are important for its users, because being able to handle a large amount of transactions in a timely manner is an important characteristic of MOM. Users usually uses MOM for message distribution in their own systems where is necessary to send and receive message in short time. For example in automated systems, where components communicates with each other by command exchange. Amount of exchanged command is dependent on system size. We want to get systems results as soon as possible and for that is important to ensure smooth and quick message exchange.

The Maestro [17] is a testing system designed especially for testing the performance of MOM. On the Figure 4.1 you can see architecture of Maestro. The whole Maestro is deployed as a cluster system on several machines. A typical Maestro deployment consist of one node for Maestro Broker, one or more for Senders, one or more for Receivers and the S.U.T. The Maestro system consists of several components:

Maestro Broker — any MQTT-capable broker with several topics. This component takes care about distribution of control messages between other cluster components such as Maestro Clients and MPT Back-end. Also, sending collected data to data server where user can browse between logs and show collected data as charts.

Meastro Clients — this component contains the client API as well as the test scripts for each test case. A sub-component called Reporter takes care of data reporting to the user, which means data visualization on the web.

MPT Back-end — consists of sender part, receiver part and inspector part. Sender and receiver ensure messaging sending to the S.U.T and receiving from it. Inspector monitoring workload over the S.U.T and reporting collected performance metrics to the testing cluster. Maestro has two back-ends:

- **Java** — using for JMS-based testing, including *AMQP* [16], OpenWire and Core protocols.
- **C** — using for AMQP and STOMP protocol testing.

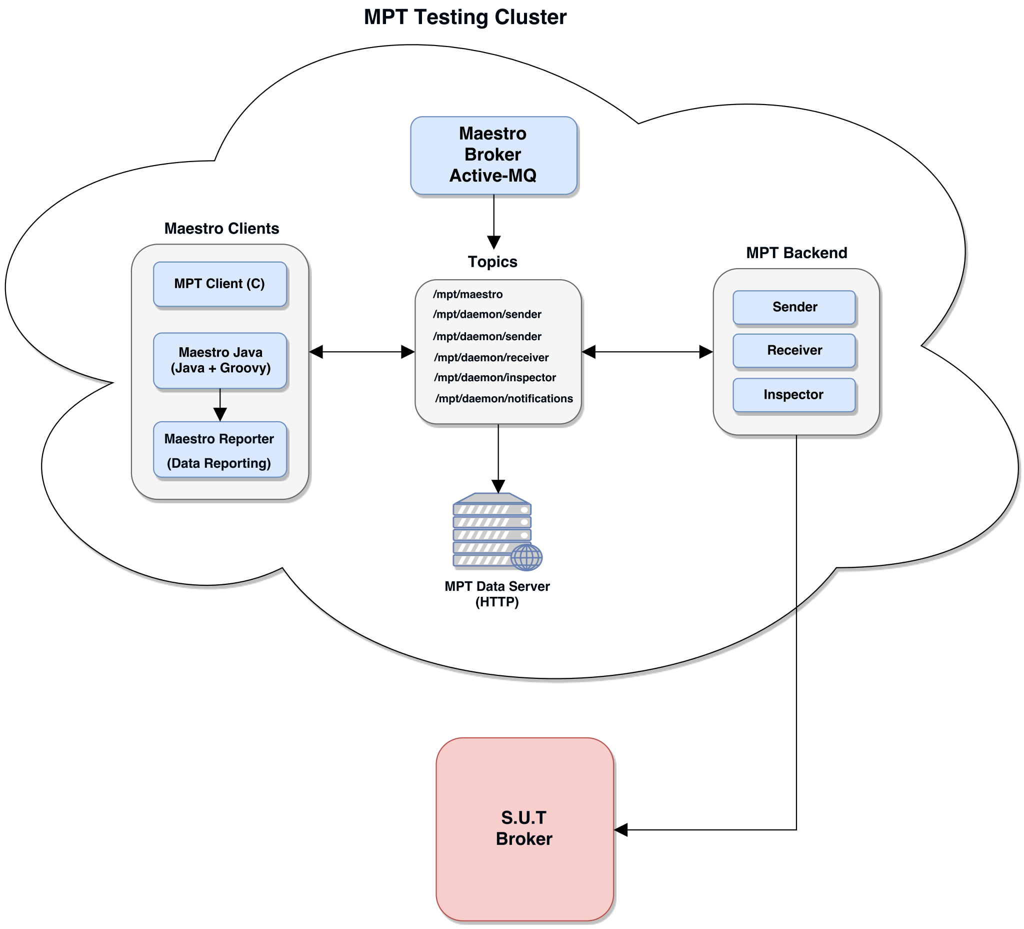

Figure 3.1: Architecture of the Maestro.

Test Case Configuration

Configuration of each test case is specified by several options defined in Groovy [1] script. This script specify test behavior with following items:

- **message size** — message size in bytes,
- **number of connected clients** — count of senders and receivers connected to the S.U.T,
- **test duration (time or load)** — end condition of each test, could be specified by time or message count,
- **message rate** — the desired rate that the system should try to maintain (0 for unbounded).

The test script is also responsible for starting and stopping the test. We can also specify the test profile. Depending on the test profile, the script may also be responsible for increasing or decreasing the workload on the S.U.T during the test scenario. The load can

[1] Object-oriented programming language for Java platform http://groovy-lang.org/

be modified by increasing the target rate or the number of parallel connections. Multiple combinations of this options can create a lot of test cases with different load for the S.U.T. Every test will produce his own logs which are processed by the reporting sub-component on the client side and used for monitoring the metrics. Maestro Reporter produce data visualization, such as charts, from stored logs.

Measures Process

Measures process starts after dynamic test generation with option passed from the test file. Sender or senders, depends on test file, will start sending messages to the S.U.T. Inspector starts monitoring the behavior of the S.U.T and report logs from S.U.T to Maestro Broker. For monitoring, Inspector uses the Broker management interface. That is a REST interface that exposes (via HTTP protocol) internal JVM [2] and Broker detailed information. That information would be normally available via JMX [3]. Data collection by Inspector is simple and pretty straightforward:

- Inspector sent an HTTP request with the JSON [4] content to the Broker REST interface.
- Broker evaluate the request and sent response to the Inspector.
- Inspector collect the response.

Errors occurred during the information collection may cause the test case fail.

However, there are two problem factors; the first is that inspector shouldn't influence performance of the S.U.T. Current method for information collecting working like the management interface call method with request for information and get response. During this call, the method usually involve locks to guarantee thread safety and exclusive access. However, calling this methods too often could cause Broker performance degradation. In order to reduce this risk, the inspector enforces a collection interval of 10 seconds and use only certain operations. That reduce the hits on management interface on 2 or 3 hits every 10 seconds.

The second is large size of the stored logs. This leads to use some of the compression methods for reduction logs size. However, compressed logs can fill whole hard drive during long test-run, so old logs has to been erase at some point of time. The collected logs can be safely erased when the test is completed. The Maestro generates about 1Gb of uncompressed data per hour of testing.

3.1 Testing Metrics

Which metrics are collected depending on the component. In the Table 3.1 we can see summary of the metrics, which are collected for each component.

[[Dopsat]]

[2] JVM - Java Virtual Machine

[3] JMX - Java Management Extensions

[4] JSON - JavaScript Object Notation https://www.json.org/

Table 3.1: Maestro metrics summary.

Component	**Metrics**	**Description**
Sender	Throughput	Throughput of the sender
Receiver	Throughput	Throughput of the receiver
	Latency	Time between send and receive message
Broker	JVM heap memory	maximum, minimum, and current Eden, Survivor, and Tenured space
	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

3.2 Collected Data and Their Evaluation

As it was mentioned, data are collected by Inspector.

[[Ask Otavio about inspector messages (format, data...)]]

3.3 Related Works

SpecJMS

4 Analysis and Design

MPT is generally designed for performance testing of Broker. However, with the service growth. the need for performance testing of Qpid-Dispatch comes. This Chapter is focused on two things: firstly, analyze Qpid-Dispatch service, describe its capabilities and methodology and secondly describe design of Topology generator and Qpid-Dispatch Performance Module for MPT.

4.1 Qpid-Dispatch Router

Qpid-Dispatch is a lightweight AMQP message router for building scalable available and performant messaging networks. With respect to ISO/OSI [1] model, this router is an application layer program running as a normal user program or as a daemon. Key features of Qpid-Dispatch are:

- 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 theory about Qpid-Dispatch router is based on knowledge available in [19].

4.1.1 Theory of Operation

The router accepts AMQP connections from clients and create AMQP connections to brokers or similar AMQP-based services. Through this connection is client (sender) able to reach message receiver, which can be another client in the network or broker. The client can exchange messages directly with another client without involving a broker at all. The router classifies incoming messages and routers them between senders and receivers. The router is designed to be deployed in topologies of multiple routers, preferably with redundant paths to provide continued connectivity in the case of 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 best path (path with the lowest cost) from sender to receiver through the whole network and to recover from failures.

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

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

4.1.2 Addresses and Connections

Qpid-Dispatch is able to connects clients servers AMQP services, and other router implementations through network connections. The router provides multiple components and settings for specify service on the other side of connection link.

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 specific address field of a source and a target. The addresses can refer to topics or queues that match multiple consumers to multiple consumers. There are two types of addresses:

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

Listener — accept client connections. Listeners has 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 to another router. Inter-router connection can only be establish over inter-route listeners.
- **route-container** — The connection is to a broker or other resource that holds known addresses.

Connector — used as an interface for create connection with brokers or other AMQP entities using connectors. Such 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** — The connection is to another router. Inter-router connection can only be establish over inter-route connectors.
- **route-container** — The connection is to a broker or other resource that holds known addresses

To ensure security the router uses *SSL/TLS - Sockets Layer and Transport Layer Security*[4]protocol and 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 connection components may be configured securely with SSL/TLS and SASL.

[5] Addresses in this discussion refer to AMQP protocol addresses, not to TCP/IP addresses.

[5] 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. This semantics control how routers behave when they see the address being used. There are two ways how the router can routing messages based on addresses:

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

- **Balanced** — An anycast method which allows multiple receivers to use the same address.
- **Closest** — An anycast method in which every message is sent along the shortest path to reach the destination.
- **Multicast** — method for send copy of the message to every receiver with the same address.

Routing mechanism — define the path to endpoint for message from sender to receiver. **[[Dopsat?]]**

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

Message may be delivered with various degrees of reliability such as *at most once*, *at least once* and *exactly once*.

4.2 Automatic Topology Generator

For various kind of messaging systems testing we needs multiple topologies with different components and setting. Manually create and deploy topology for each test scenario is slow and annoying, after few scenarios. Solution of this problem is divided into two parts: simple topology generator, that transform metadata, defined by user, into configuration files for each component contained in metadata, and automatic *Ansible* scripts, which deploy whole topology to physical machines. User has to define metadata file, single file for whole topology instead of single file for each component, and then start Ansible script which ensure configuration files generation and deployment.

4.2.1 Topology Components

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

Broker

Broker configuration file offers various setting and protocols such as specialized queueing behaviors, message persistence, and manageability. The following itemize shows several capabilities of broker:

- **User access** — allows guest access or authentication access for 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** — refer to approach how to deliver messages, examples are point-to-point and publish-subscribe approach.
- **Logging level** — broker offers to setup different logging levels.

However, broker configuration is not implemented yet, but design of 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 I recommend Qpid-Dispatch documentation [19].

4.2.2 Format of Input and Output

Input structure should be user-friendly and easy to update even in case of large topologies. As input is choose one file called `config.yml` in *YAML*[6]language, which is same as JSON but its better readable for humans. Generator needs information about all hosts in topology and which type of topology it should generate. For that purpose is there two attributes in configuration file; first is *inventory path* which refer to location of *Inventory*. Inventory is file, containing all hosts in topology in specific format. This format is available in A.1. It is simple configuration file with enumeration of host names and their IP addresses. Second attribute is which type of topology it should generate. User can specify there one of the simple types of graph, such as line, circle, complete, etc., which do not need any other information except Inventory or he can specify path to graph metadata. Graph metadata are deeply described in Subsection 4.2.3.

On the other hand, output format should be easy for automatic parsing. The best format for machine parsing is JSON or YAML format, since both of them are loaded with same functions in ansible. Output of the generator will be pass to ansible script immediately after create without any user intervention. However, user should have option to see generator output in YAML format, because in case of larger topologies JSON is badly readable for human eyes. 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.

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

4.2.3 Graph Metadata

At first, we should specify technology which will be used for implementation of Topology Generator. *NetworkX* is a Python package for creation and manipulation of complex networks. This package offer features for creating graphs, multigraphs, random graph generators, plot created graph, and etc. NetworkX also offers graph import and export in YAML structured file. This type of file is very useful as graph metadata, simple example of this file is show at A.2.

User can specify here any setting for each node. For example, user can specify listener for router1 and connector for router2 as you can see on 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 create two nodes with type, id, and listener or connector attributes. This attributes will be used for generate configuration files for each node. All possible attributes that user can specify for each node is available in A.4.

However, specify all attributes of each node is not very user-friendly approach, especially in case of large topologies. User could only specify nodes and links between them and generator will add all necessary attributes for establish connection between nodes. Example of this metadata file you can see in A.2.

4.3 Topology Deployment

Every node specified in Inventory has to receive proper configuration files for services running on it. This job will be handle by 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 create configuration files based on loaded variables and common template for Qpid-Dispatch. Created file will be sent to proper node based on node name from Inventory, which has to be same like router name specified in generated variables.

4.4 Qpid-Dispatch Performance Module

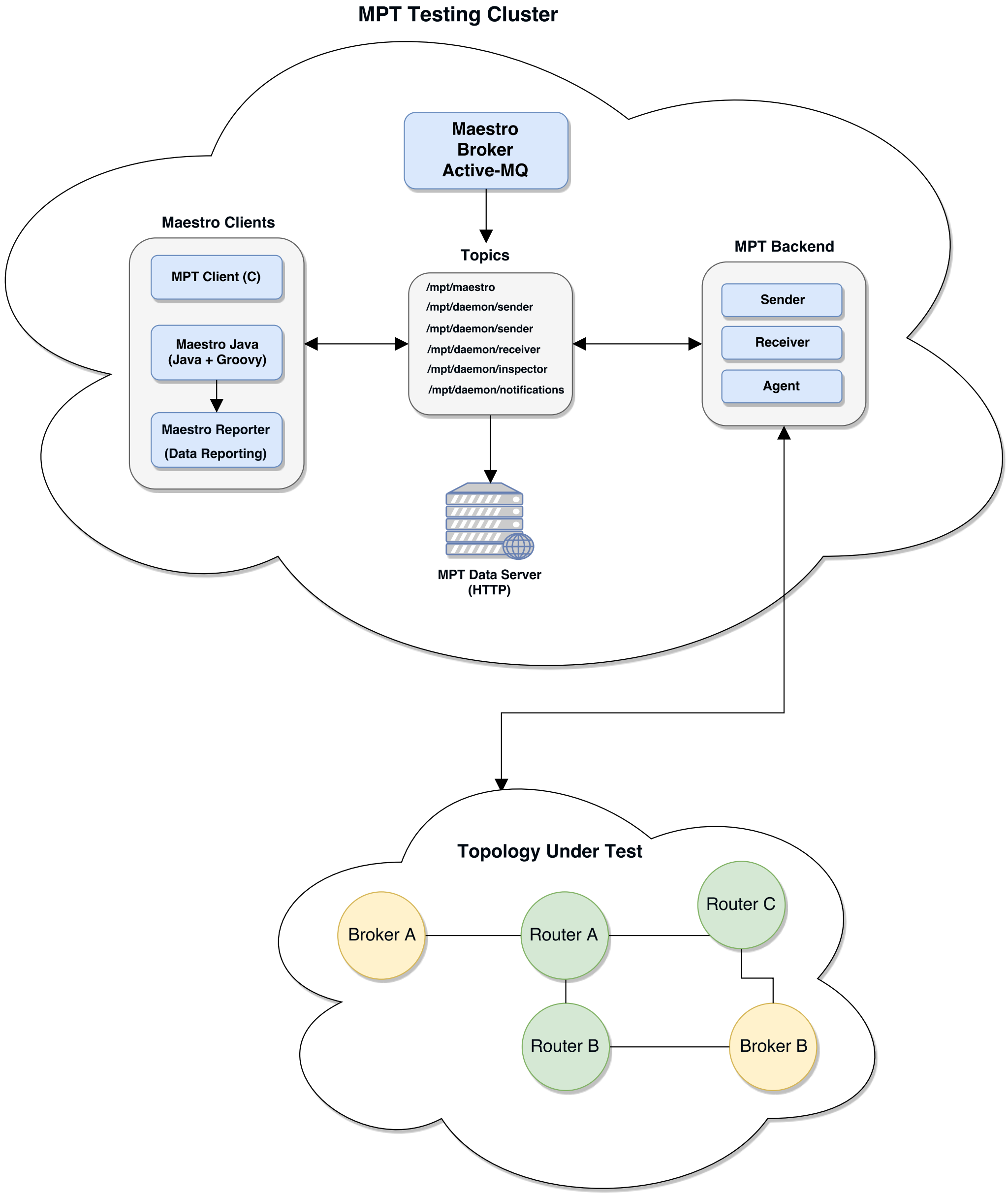

Figure 4.1: Architecture of updated Maestro for testing of Qpid-dispatch router.

4.4.1 [[more subsections about module]]

4.5 Performance and Testing Metrics of Qpid-Dispatch Performance Module

4.6 Collected Data Evaluation

5 Implementation

5.1 Used Technologies

5.1.1 Ansible

5.1.2 Docker

Using for testing Ansible roles (remove?)

5.2 Topology Generator

5.2.1 Template Generator

5.2.2 Generation of Variables

5.2.3 Configuration Files Generation and Deployment

5.3 Qpid-Dispatch Performance Module

5.3.1 TODO - more subsections about implementation

6 Experimental Evaluation

6.1 Performance Testing on Various Generated Topology

6.2 Testing results

7 Future work and ideas

8 Summary

Bibliography

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

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

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

[4] 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

[5] 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/

[6] 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

[7] 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

[8] 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

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

[10] 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

[11] 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/

[12] 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

[13] 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

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

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

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

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

[18] 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.

[19] 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.

[20] 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 Configuration Files

A.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
```

A.2 Graph Metadata

An 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
```

A.3 Topology Generator Output

Path to output file?

A.4 Qpid-Dispatch Configuration File Template

Path to template file?