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

AUTOR **Bc. JAKUB STEJSKAL**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ FIEDOR**
VEDOUCÍ PRÁCE

BRNO 2018

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 and it analyzes performance testing of components used in Messaging systems, especially Message Broker and Qpid-Dispatch. However, currently used methods for performance testing of these components are primarily focused only on Messaging Broker by system Messaging Performance Tool called Maestro. This paper proposes improvements of Messaging Performance Tool to allow proper performance testing of Qpid-Dispatch and its capabilities in automatic testing. The solution is demonstrated on series of experiments with different topologies. 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 všeho druhu. Tato práce mapuje základy testování výkonu, které jsou aplikovatelné na libovolné aplikace a následně analýzuje testování výkonu komponent používaných v Messaging systémech a to konkrétně Message Broker a Qpid-Dispatch. Využívané metody testování výkonu je zaměřeno zejména na Message Broker pomocí systému Messaging Performance Tool s názvem Maestro. Práce navrhuje vylepšení této aplikace o rozšíření testování systému Qpid-Dispatch a její možnosti při automatizovaném testování. Řešení je demonstrováno na sérii experimentů s různými topologiemi. Výsledná zpráva závěrem vyhodnocuje navržené rozšíření systému Maestro, zhodnocuje výkon komponenty Qpid-Dispatch a rozvíjí myšlenky pro další rozšíření.

Keywords

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

Klíčová slova

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

Reference

STEJSKAL, Jakub. *Performance Testing and Analysis of Qpid Dispatch Router*. Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Fiedor

Rozšířený abstrakt

Jedním z hlavních cílů během vývoje softwaru je přijatelný výkon vytvořené aplikace. Mimořádný důraz na výkonost softwaru je pak hlavně kladen například na aplikace používané ve vesmírných programech, zdravotnictví, armádních systémech a nebo v systémech pro distribuci energie. V těchto odvětvích je nutno garantovat správné chování aplikace po neomezenou dobu běhu pod vysokou zátěží, a to bez viditelných výkonostních problémů jako je vysoká doba odezvy, častá zpoždění nebo vypršení časových limitů pro spojení. Protože sebemenší chyba pak může mít fatální následky.

Současně je ale v dnešních dnech vyžadován i hladký běh síťových aplikací a systémů a to hlavně kvůli stále frekventovanější komunikaci přes internet. Pro internetovou komunikaci zpravidla využíváme různé komponenty jako jsou hardwarové směrovače či přepínače, ale také softwarové verze těchto komponent spojené do tzv. *Messaging systémů*. Příkladem součásti těchto systémů je komponenta *Message Broker* — distributor zpráv v síti — nebo *Qpid-dispatch* — směrovač na aplikační vrstvě. Obě komponenty jsou vyvíjeny společností Red Hat Inc. a k jejich výkonostnímu testování se používá nástroj Maestro.

Hlavním přínosem této práce je tímto rozšíření zmíněného nástroje pro výkoností testování Maestro, který se zaměřuje na výkonostní testování Messaging systémů (Message-oriented middleware), s důrazným zaměřením na komponentu Messaging Broker. Práce popisuje zejména architekturu celého systému a komunikaci mezi jednotlivými komponentami pomocí MQTT protokolu. Aby bylo možné využívat nástroj Maestro pro testování Qpid-Dispatch efektivně a s možností simulovat reálný provoz, bylo dále nutné navrhnout a realizovat nové komponenty pro Maestra, které tento druh testování umožnily. Těmito komponentami je *Maestro Agent*, který umožňuje za běhu testu vyvolat externí události v síti, a *AMQP Inspector*, který umožňuje kontinuální monitorování právě Qpid-Dispatch, například pro sledování počtu připojení, velikosti alokované paměti nebo počtu přenesených zpráv. Implementace těchto komponent ale vyžadovala zásahy do komunikačního systému Maestra.

Součástí implementace je také navržení a realizace externího nástroje pro generování a následné nahrání topologií skládajících se z Qpid-Dispatch uzlů. Tento nástroj umožňuje na základě metadat vytvořit konfigurační soubory pro všechny Qpid-Dispatch uzly v síti a pomocí nástroje Ansible jsme schopni tyto soubory jednoduše a automatizovaně nahrát na cílové stroje, čímž lze relativně snadno topologii měnit například mezi různými testy.

Realizovaná implementace byla experimentálně ověřena na sadě příkladů s různými topologiemi. Díky integračnímu nástroji Jenkins bylo rovněž možné provádět plně automatizované testování, včetně změn topologie. Testování probíhalo na strojích v laboratoři s operačním systémem Red Hat Enterprise Linux a nainstalovanými komponentami Qpid-Dispatch, případně Messaging Broker. Experimenty byly prováděny s verzí Maestra 1.3.0, kde byly zakomponovány rozšíření Maestro Agent a AMQP Inspector. Naměřené výsledky ukazují řadu zajímavých faktů, jako je například přílišná degradace propustnosti linky při topologii sériového zapojení několika Qpid-Dispatch routerů. Zdrojové kódy jsou zveřejněny jako open-source a jsou dostupné na GitHubu. Navržené a implementované rozšíření je již reálně nasazené a používá se k výkonostnímu testování nových verzí komponent Messaging Broker a Qpid-dispatch.

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 were 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
May 17, 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 guidance and providing important insight about performance problems. Also I would like to thank my colleagues Otavio Rodolfo Piske for his time during the introduction and explanation of Maestro and help with the development and CI integration, and Dominik Lenoch for introduction to Qpid-Dispatch service.

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 still be a lot of issues, which can badly influence the 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 have to 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 a 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 already handle complex underlying issues such as security, performance, or 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 the performance which naturally 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 or are too expensive.

A very important part of the performance analysis is the right choice of so called *key performance indicators* (KPIs) [19] 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* [19] 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 performed 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 the 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 systems developed by *Red Hat Inc.* there is an existing solution — Messaging Performance Tool (MPT) called *Maestro* [21].

MPT is specialized for the performance testing of *Message Broker* (Broker) [22] — network application level software cooperating with *Qpid-dispatch service* [23] in the network as the message distributor. Unfortunately, the current version of Maestro does not support performance testing of enough components 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 Maestro described in Chapter 3. Description includes *measurement process* and *measured data description and evaluation.*The main goal of the thesis is to analyze Maestro 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

The usual goal of the 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 has slow response, long load time or bad scalability, the first website which user will visit afterwards will be the 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 the performance testing which includes definitions of common performance processes, issues, and metrics, based on knowledge available in [19, 18, 12, 2].

2.1 Performance Testing Process

The main goal of the performance testing is to ensure the following application attributes [14]:

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 amounts of workload (such as network traffic, server load, data transfer, etc.) which would need new hardware for cluster expansion,

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.

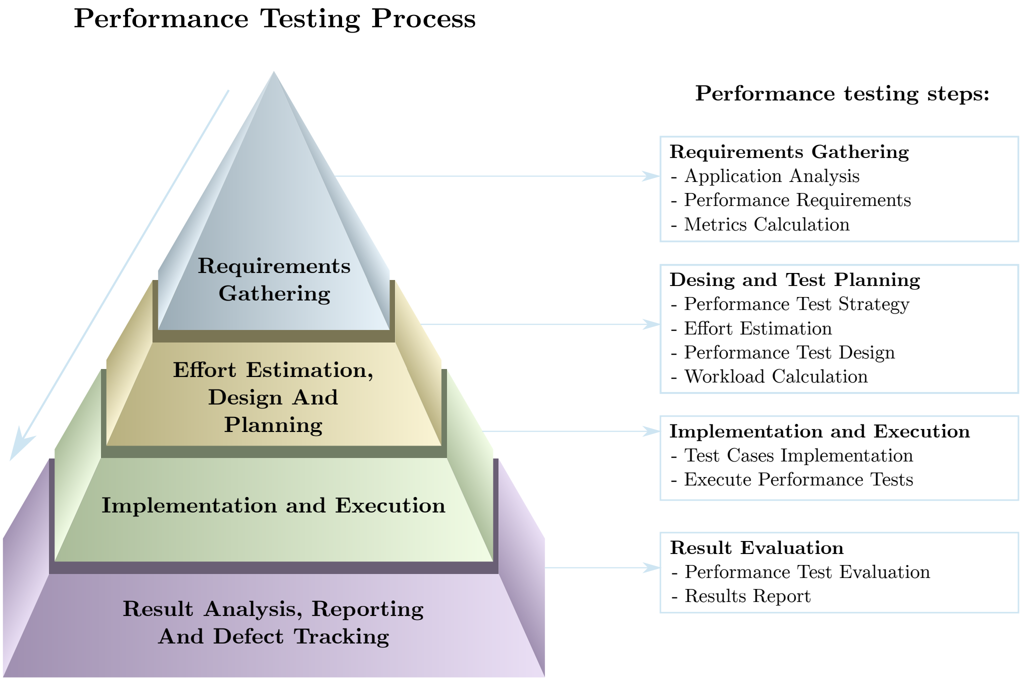

Performance Testing Process
Requirements Gathering
Effort Estimation, Design And Planning
Implementation and Execution
Result Analysis, Reporting And Defect Tracking
Performance testing steps:
Requirements Gathering
- Application Analysis
- Performance Requirements
- Metrics Calculation
Desing and Test 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: The performance testing process with the four most important parts and theirs individual steps based on [24].

In the Figure 2.1 you can see the scheme of performance testing process where each level represent required time for each step. Lower levels refers 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*, chose suitable performance metrics, that will model the application performance, and state 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 six months or in one 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 six months and one 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 such as 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 w.r.t this document.

The next step is to define the *performance testing strategy*, corresponding to the 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 includes choosing the right hardware for the testing, then installing the necessary software for running load injectors, tested components, etc., and preparing other equipment depending on the application purpose such as routers, switches, mobile devices, etc.

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

Identify and implement use cases — this includes 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 — tests can detect significant performance issues, but their investigation and the actual fixes may take a long time. After the fix the retest of issue is needed.

While this process seems trivial, the opposite is true, especially in cases of network applications. Most of the performance issues manifest with big workloads or high number of users, e.g. when million users are sending requests to the network device at the same time it can 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 the execution, so the 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.

The final step of the performance testing process is evaluation of the results. 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

A *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 the following, we will describe selected issues in more details.

Performance Degradation

An unclean code usually leads to inefficient algorithms, application deadlocks, or memory leaks, which all can eventually cause a 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 the *endurance testing* [9, 16] which is described in Section 2.3. The endurance test is intended to identify problems that may appear only after the long period of the application run-time[2], hence its necessary to run this type of tests during the application development. The network applications usually need to be available for 24 hours per day. The duration of a endurance test should have some correlation to the operational mode of the system under test. Following scenarios may represent performance issues detectable by endurance 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 the 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 the long run time such as a backup process, exporting of data to a 3rd party system, etc.,
- a development of new features for already existing components.

Response Time

Response time represents how long it takes for system to accept, evaluate, and respond to the user for his request e.g. HTTP request for the 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 a 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* [16] 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 three 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 the application.

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

However response time thresholds for non-human interactive system are more strict. They can range in milliseconds or less.

Traffic Spikes

As a *traffic spike* [18, 6] we can understand the sudden surge in demand from users. Typically manifesting by doubling or multiplying of traffic level in a short period of time. In a 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 a sudden traffic spike in response time after publishing new popular viral content on video servers, start of sales events, reservation of limited amount tickets or subject registration at university. Scheduled automatic backup or system upgrade for whole company during early morning hours can also cause traffic spikes.

Traffic spikes can lead to the 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 a sophisticated infrastructure monitoring and network load analysis, in order to distinguish between normal traffic and an attack on the system. Suitable methods for testing of spikes is one of variant of *stress testing* [16] 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.

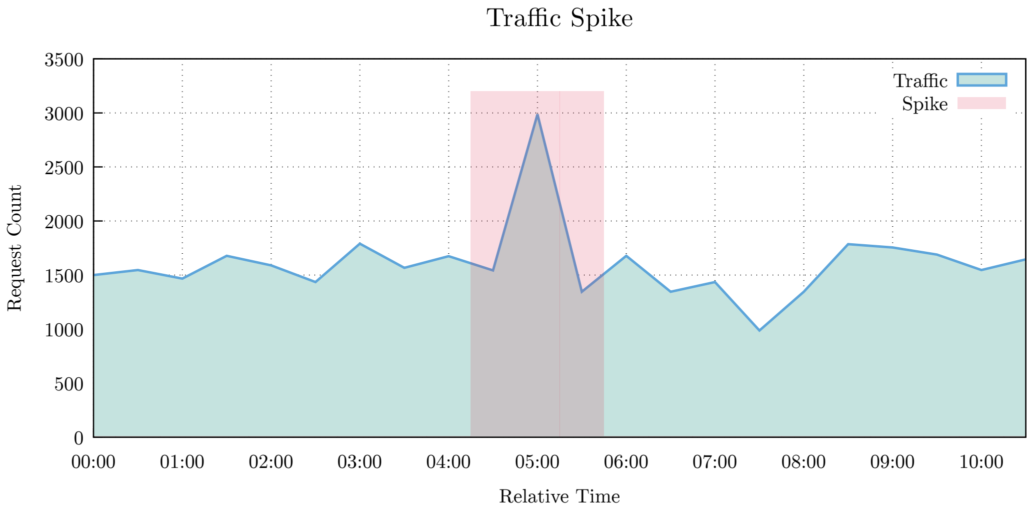


Figure 2.2: The graph shows amount of concurrent sessions depending on time. During to network traffic monitoring we can see the traffic spike occurring after five hours from test start.

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 a category or suite of the tests:

- **Testing methods** — load testing, stress testing, endurance testing,
- **Testing approaches** — smoke testing, regression testing, benchmark testing.

Their description is based on the knowledge available in [5, 9, 19, 2].

Load Testing

Finding the 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 response time metric of the system for requests. Requests are generated by users or another systems communicating with the SUT. The 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 has reached its maximum operating capacity. After the successful testing, we should mark the workload requirements as fulfilling or analyze the 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.

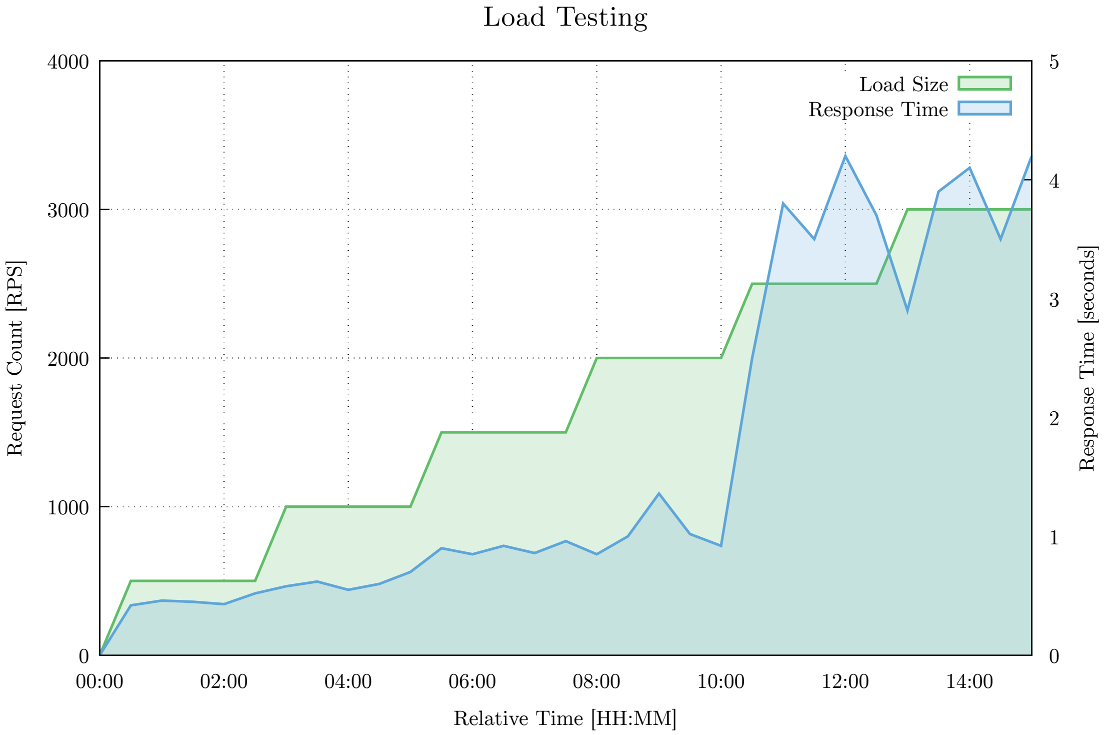


Figure 2.3: The response time of the system during the load testing depended on load size.

The following list shows common scenarios for load testing:

- The system interacting with multiple users at same time.
- The system tracking communication and analyzing it.

- Web services and information systems.

Typical system issues covered by the load testing:

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

Stress Testing

Stress testing is the specific type of load testing, where we do not measure the 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 an 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:

- Monitoring 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 having a server in a cluster down.

Typical issues, which are covered by stress testing are as follows:

- A sudden performance degradation.
- System will recover after the 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 finds the limits of the system, they also can test the system recovery after a crash during finding of the system limits.

In the Figure 2.4 we show recorded stress testing with a 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.

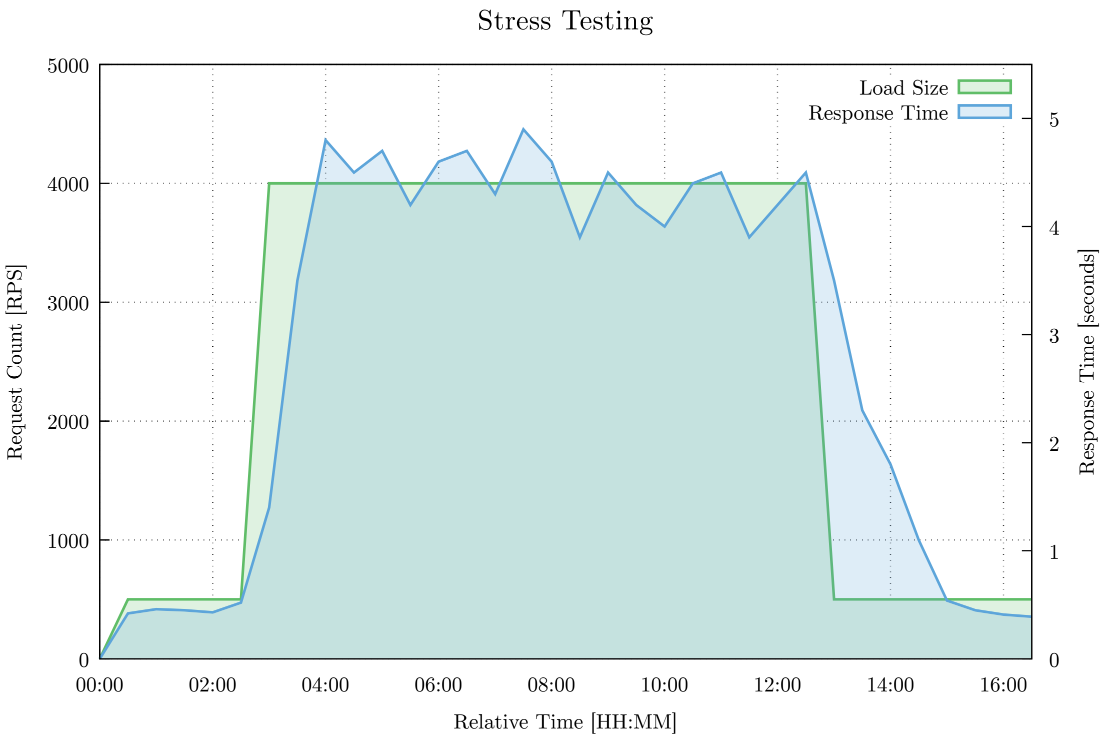


Figure 2.4: Stress testing diagram capturing dependency of response time on amount of requests.

Endurance Testing

The endurance, or stability/soak testing refers to the method, that tries to identify problems, that may appear only after the extended period of time e.g.. The system could seem to be 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 the memory as a performance metric. The following are common issues found by soak test:

- A 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 a high response time and the system becomes inefficient.

Typical scenarios for usage of soak testing:

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

This sort of test needs to use appropriate monitoring system to achieve the 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.

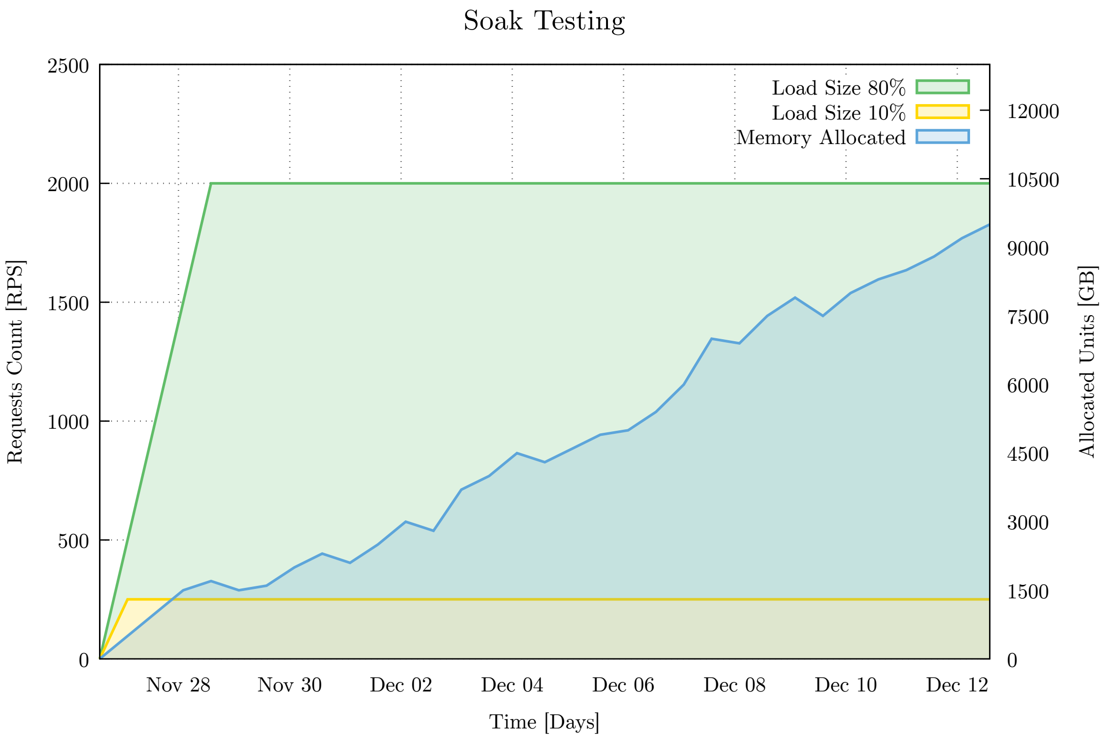


Figure 2.5: Soak testing with memory usage dependent on time.

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 and so the SUT will crash. This may have been caused by a memory leak or an inappropriate algorithm use.

Smoke Testing

The smoke testing approach is inspired by the similar hardware technique, when engineers checks for the presence of the smoke from the device after turning the power on. Basically, its similar for software, since the main goal of smoke test is to test the basic functionality of the system and guarantee that the system is ready for the build. However, smoke tests are testing the functionality on a surface level, so they may not be enough for the 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 resulting into a system crash.

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

Regression Testing

Whenever engineers develop a 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 the 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

The *benchmark testing*[3] is an approach, which collects performance data during the system run on different hardware machines [17]. Collected Data Format has significant value when we want smooth run of the system on an older hardware, hence we can discover performance issues under normal load. However, when the system does not run smoothly on prepared hardware, the only option 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, not depending on developing language.

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

They Ways to Measure

The 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

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

unstable workload can negatively affect the measurement results. In the Figure 2.6 one can see a workload phases with marked part for the actual performance testing.

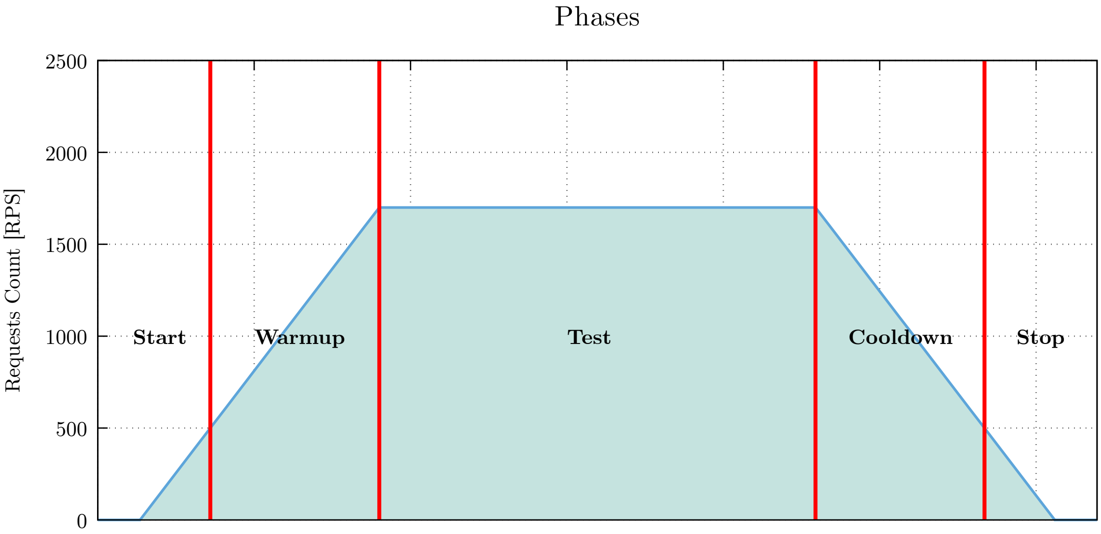


Figure 2.6: Load phases of performance measurement process.

Workload during testing does not have to be on the same during the whole testing. In particular, load testing finds the highest load during which the system can work properly. This limit is found by raising the load and monitoring 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.

2.4.2 Response Time and Latency

Slow response time as an issue was already mentioned in the Subsection 2.2; response time as metric consists of two parts — *latency* and *service time*.

Service Time

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

Latency

The second part of the response time is the latency [8, 7], which represents a delay between the 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 centers, web servers, etc., because request/response needs to travel over the physical medium between the client and the server. Client and server can be located on different continents, thus the message has to travel long distance and the latency increases.

Round Trip Time

Round-trip time (RTT) is a time that it takes for a signal to be sent together with a 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 depends on the distance between the sender and receiver, because that is the distance the signals must travel by.

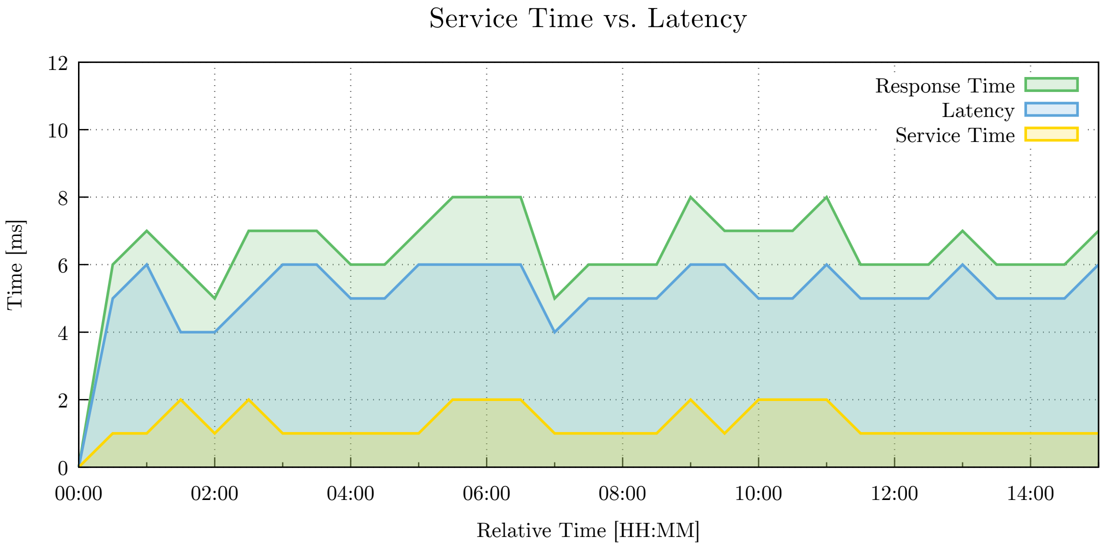


Figure 2.7: Diagram capturing the difference between the latency and response time.

In the Figure 2.7 you can see the response time and both of it's parts: latency and service time. Service time is usually smaller than latency since latency depends on the 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 [15]: one of them being the average (mean) response time 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 applications have few heavy outliers such as several 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.

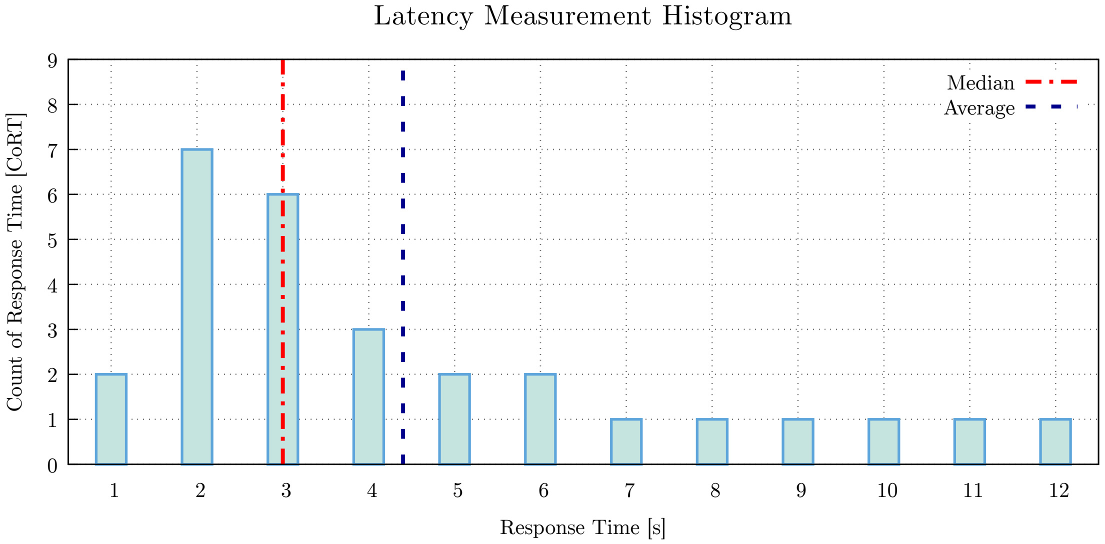


Figure 2.8: Transactions response time with calculated average and median of response time. The average represent inaccurate response time, which is higher than real one.

The better solution how to determine the actual response time is the Percentile. The percentile is statistic method, which cuts 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.

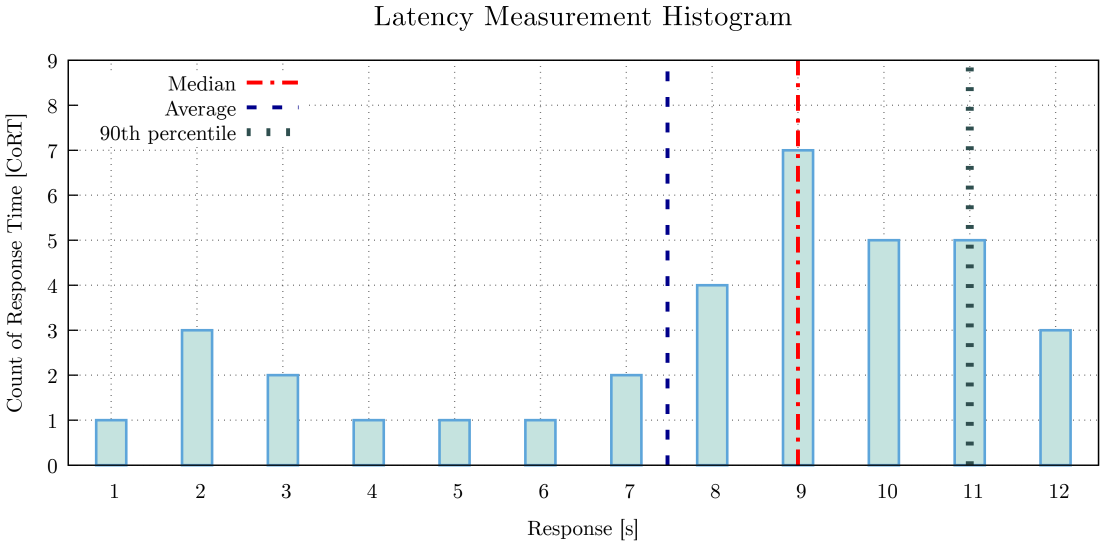


Figure 2.9: Transactions response time with calculated average and median of response time.

The Figure 2.9 shows a different situation. The average represent inaccurate response time, which says, that SUT is faster than it is in a 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, the calculated percentile gets more realistic value than average response time.

2.4.3 Resource Usage

Applications running at servers with long run-time competes over a limited amount of resources 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 a zero network errors present, however, in reality this is infeasible. 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 the problem easily.

3 Messaging Performance Tool

The 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 the term the messaging client to just client in this thesis.

The 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. The 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 (or Messaging Performance Tool) [21] 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 Maestro Senders, and one or more for Maestro 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 a queue with a name where other messaging services can listen on the traffic. This component takes care of 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 visualizations.

MPT Back-end — consists of sender, receiver and inspector parts. Sender and receiver handle 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)* [20], OpenWire and Core protocols.
- **C** — used for AMQP and *Streaming Text Oriented Messaging Protocol*[3] *(STOMP)* protocol testing.

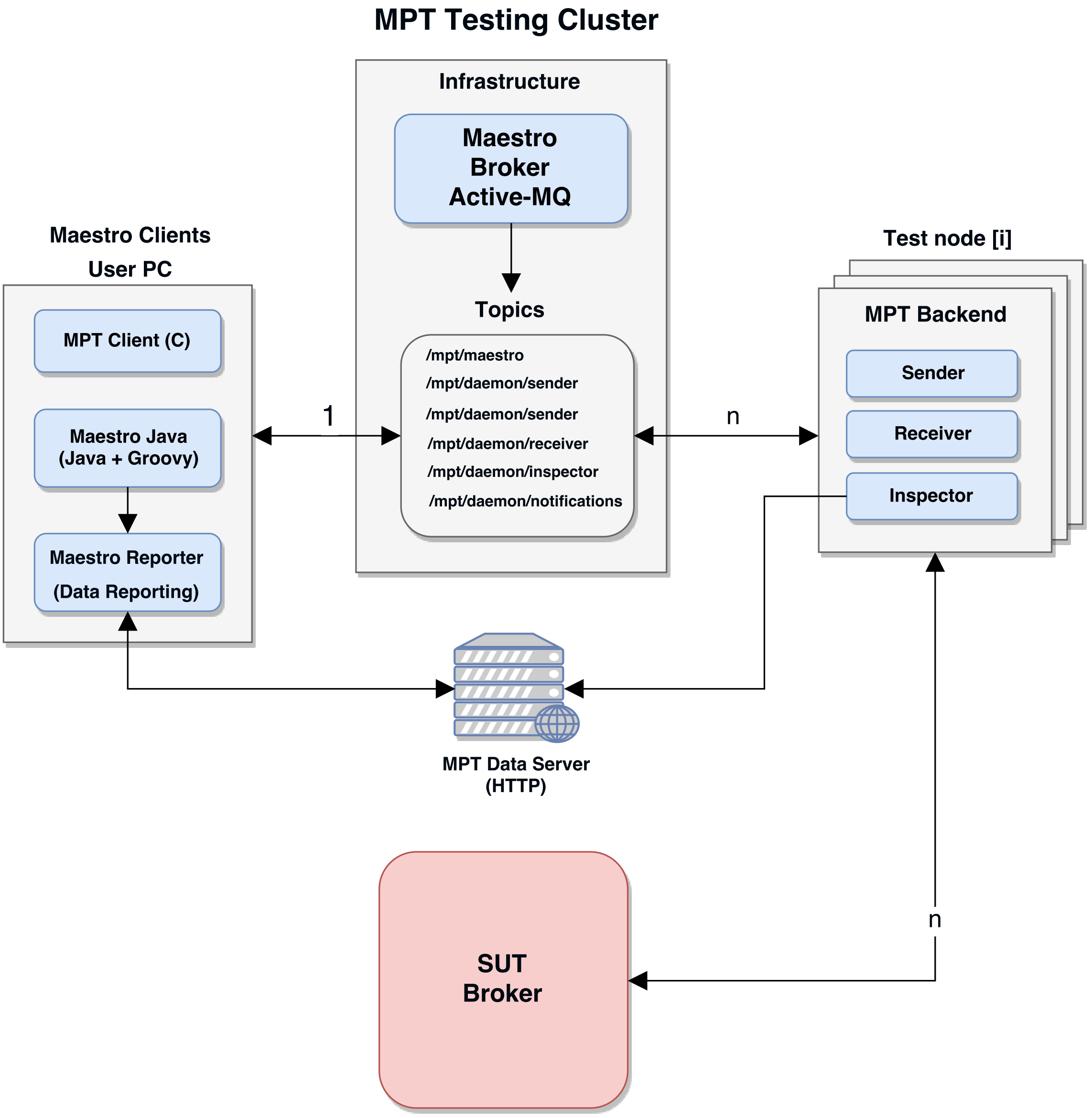


Figure 3.1: The architecture of the Maestro. The Maestro contains Maestro Clients as a front-end; Maestro Broker as a message distributor; and sender, receiver and inspectors as a backend.

[2] JMS — Java Message Service

[3] STOMP — https://stomp.github.io/

3.1 Test Case Scenario

The test is basically a generation of huge amount of messages followed by sending them to SUT and then receiving them. The configuration of each test case is specified by several options defined in the 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** — the count of senders and receivers connected to the SUT,
- **test duration (time or load)** — the 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 (0 for unbounded rate).

The test script is also responsible for starting and stopping the test. Moreover the test case can be extended by the 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 produces data visualizations, such as the test overview and charts (rate based on time and 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 the MessagePack[5]. For the message exchange between nodes it currently uses MQTT protocol (version 3.1.1) and for sending the testing data to the 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, and 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 commands 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 sends 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 solution 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 a 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 a 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. 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

While Maestro is relatively new system, there are only few existing performance testing tools for MOM. Noteworthy are two tools, which were used for performance testing before the maestro development. These tools are *SpecJMS* [10] and *JMeter*[10], the advantages and disadvantages are described in the following.

SpecJMS

SpecJMS is the industry-standard benchmark for evaluating the performance of enterprise message-oriented middlevare servers based on JMS. Basically, SpecJMS runs real-world scenarios, which simulate real load over the messaging topology. SpecJMS collects data during the test and then evaluates it as a score. This score is a standardized value, which

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

[10] JMeter — https://jmeter.apache.org/

represent a performance of the tested system. Each system tested by SpecJMS can be compared with another system based on the computed score. Note, that a fair comparison between a tested systems involves run the tests on the same hardware.

The great advantage of SpecJMS is the comparison between the different tested systems only based on the performance score. However, it has a poor test case capabilities, since the test cases are pre-defined by the SpecJMS developers and designed only for JMS. Nowadays, this benchmark tool is retired and is no longer supported.

JMeter

The Apache JMeter is an open source software designed to load test the functional behavior and measure performance. JMeter system is basically an IDE written in Java, which offers a performance testing of web applications, servers and MOM (via JMS only) by a simulation of a heavy load. JMeter testing script capabilities are better then SpecJMS has. Also the JMS restriction for MOM is not very comfortable, since Qpid-dispatch can handle more than only JMS connections such as Qpid-proton, Ruby or any connection type which is able to use the AMQP protocol. The different connection type during the test can be tested by Maestro as well. Maestro also implements interior data collection about the router itself, which is very useful during the performance bug hunt.

4 Analysis and Design

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

4.1 Used Technologies

The most of Maestro, such as the command parsing, reporting, clients abstractions and so on, is written in Java language. But the whole Maestro is not a pure Java code. For test specificatio we use Groovy instead. Groovy is basically a lightweight version of Java with several advantages. In particular, Groovy scripts are more readable for those who are not much familiar with Java code. Groovy scrips are also used as handlers for specific commands for extension points, which is described in more depth 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 it was developed in the *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. The rest of the following will describe the rest of the used technologies.

4.1.1 Ansible

Ansible [3] is a 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 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, while Windows requires PowerShell; both systems requires SSH support. Moreover, Ansible playbooks can be grouped together and create more complex scripts called roles. These are open-source and available in the public repository.

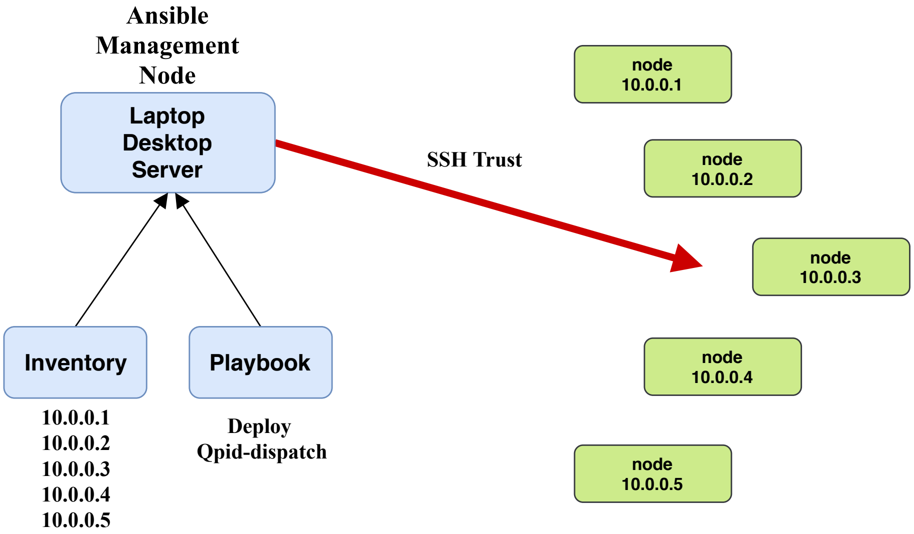


Figure 4.1: Example of Ansible architecture with several nodes. Inventory and Playbook are passed to Ansible Management node, which executes the playbook on all node specified in the inventory.

We use Ansible for several tasks; mainly to deploy systems on specific nodes. As we 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 Maestro, Qpid-dispatch and other services based on the testing scenario. The next usage is to create and deploy configuration files for each router. This task runs the Topology generator and creates configuration files for each machine based on the generator output.

4.1.2 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 the 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 containers are build up from a dockerfile where container attributes are specified such as its 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.

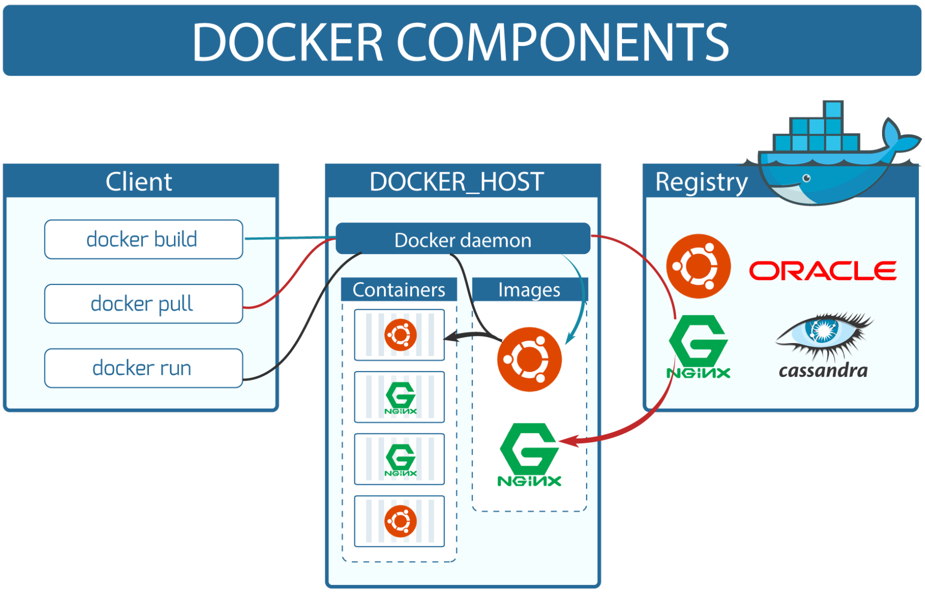


Figure 4.2: Docker architecture with all its components and commands. Docker can pull or build specific image and then run it in docker container.

Since docker is able to run services such as Qpid-dispatch very easily and also allows communication between containers, it is possible to deploy Maestro with proper SUT in containers and analyze behavior in the container network or just run Maestro on single machine. However, for proper performance results we need real machines, so docker containers we used only for Maestro development and trying some basic stuffs with Maestro. The docker architecture is depicted in the Figure 4.2 [13].

4.2 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, w.r.t. 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 [23].

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

4.2.1 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. Note, that the client can exchange messages directly with another client without involving a broker at all. The router classifies all of the 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 continually provide 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 path with the lowest cost) from sender to receiver through the whole network and to recover from failures.

4.2.2 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 then a message distributor.
- **link route** — the address is a private messaging path between sender and receiver. The router than only passes messages between end points.

Listener — is used to accept 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 only link 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 a known address.

Connector — is used as an interface for creating a 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.

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

- **inter-router** and **route-container** — they are the same as listener's modes.

To ensure the security the router uses the *SSL/TLS (Sockets Layer and Transport Layer Security)*[4] protocol and its 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.

4.2.3 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 — defines paths that message with a mobile address can take. These routing patterns can be used in both cases 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 — defines 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*.

4.3 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 with just a few scenarios.

[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

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

The solution to this problem is divided into two parts: a simple topology generator, which 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 only 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.

4.3.1 Topology Components

Messaging system consists of multiple components with specific roles. In our case, testing topologies will contain 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 Maestro deals with it as was mentioned at Chapter 3.

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 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, e.g. are point-to-point or publish-subscribe approach.
- **Logging level** — broker offers the setup for different logging levels.

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

Router

Similarly to the broker configuration, the router offers various types of configurations. The basics were explained in Subsections 4.2.2 and 4.2.3, but for better understanding of all capabilities we recommend to refer to Qpid-Dispatch documentation [23].

4.3.2 Input and Output Format

The input format should be user-friendly and easy to update even for large topologies. Hence, as the input we choose one single file (`config.yml`) in *YAML*[7] language, which is similar to JSON format but 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; the first is the *inventory path* which refers to the location of *Inventory* — file, containing all hosts in topology in the specific format (for its specification refer to Appendix B.1). It is a simple configuration file with enumeration of host names and their IP addresses. The other attribute is the type of the topology it should generate. The user can either specify one of the simple types of graph, such as line, circle, or complete, which does not need any other information except Inventory or one can specify path to graph metadata, which are described in Subsection 4.3.3 in more details.

On the other hand, the 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 can be 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. 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 4.3.

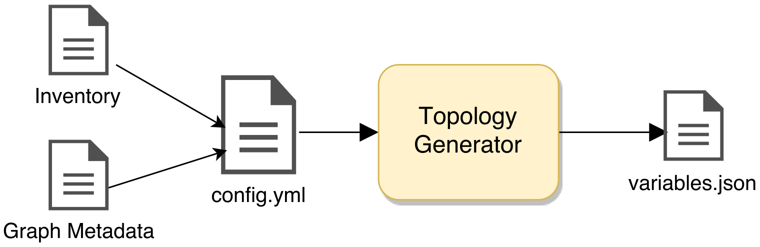


Figure 4.3: Topology generator takes input YAML configuration containing specification of graph metadata and outputs sets of variables in JSON format.

4.3.3 Graph Metadata

The technology 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 a graph metadata; simple example of file is shown in Appendix B.2.

In these 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
```

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

```
- type: router
  id: router2
  connector:
   - name: router1
     host: router1
     port: 5675
     role: inter-router
multigraph: false
```

From these metadata NetworkX creates two nodes with type, id, and listener or connector attributes. These attributes will be used to generate 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 the case of large topologies. So user can only specify nodes and links between them and generator will add all necessary default attributes in order to establish connection between nodes. The example of this metadata file can be seen in Appendix B.2.

4.3.4 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 loads 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 as router name specified in generated variables. The scheme of configuration deployment is depicted in the Figure 4.4.

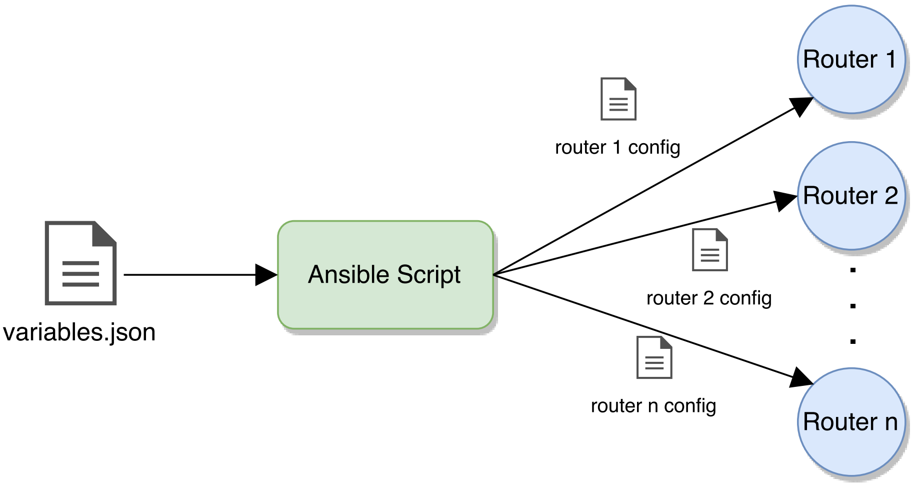


Figure 4.4: The scheme of configuration files deployment to the nodes The Ansible script takes input file with variables generated by Topology Generator, fills the configurations template and deploy them to corresponding nodes.

4.4 Agent Performance Module

The architecture of Maestro (as depicted in the Figure 3.1) originally could not use all performance testing and network recovery possibilities of the Qpid-Dispatch. Hence, for better performance analysis and measurements it was necessary to design and implement additional functionality for Maestro.

In the Figure 4.5 we show updated version of Maestro architecture. Proper performance testing of router and network analysis with few routers needs special agent, which can manipulate each node. In particular, Maestro 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*.

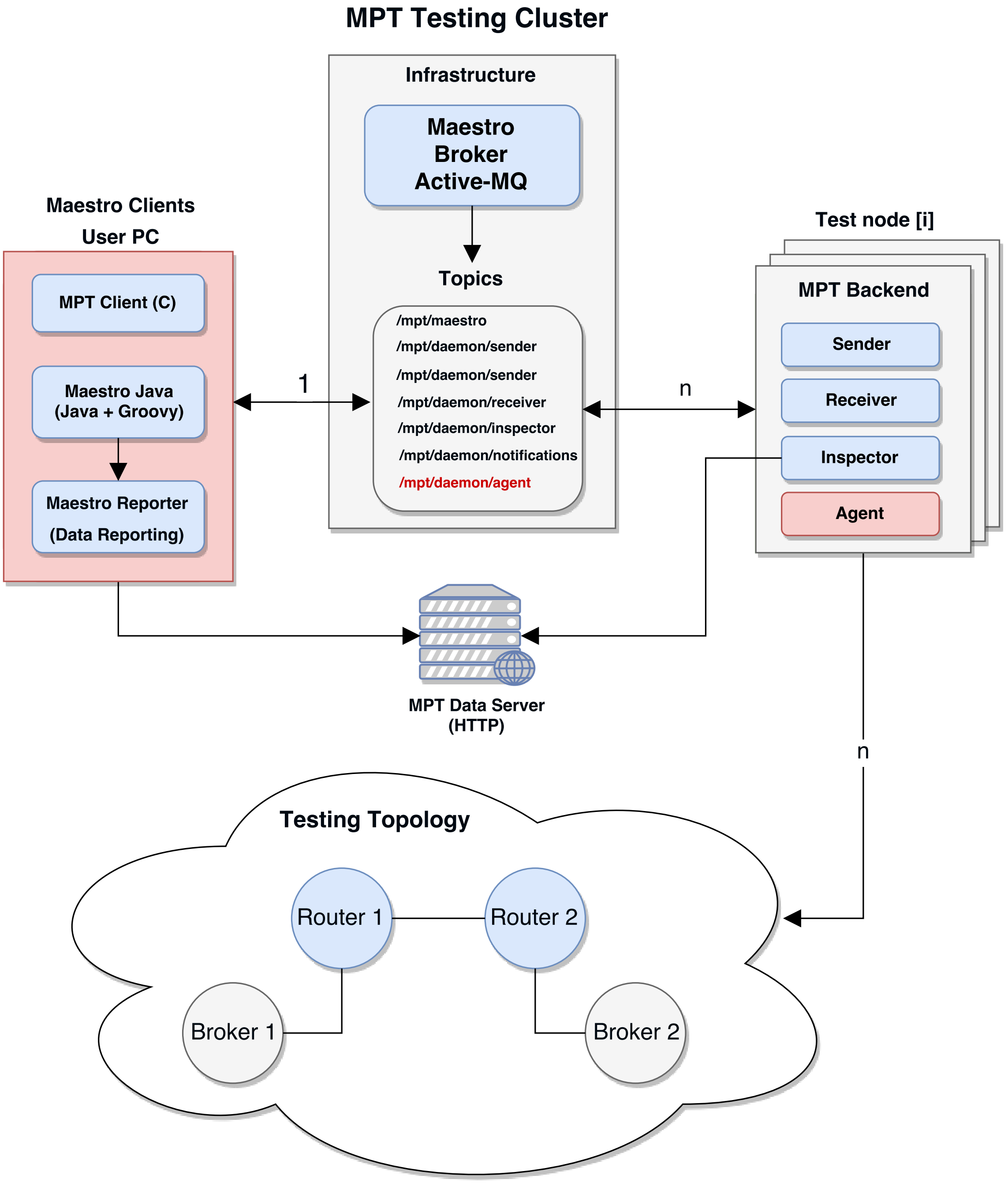
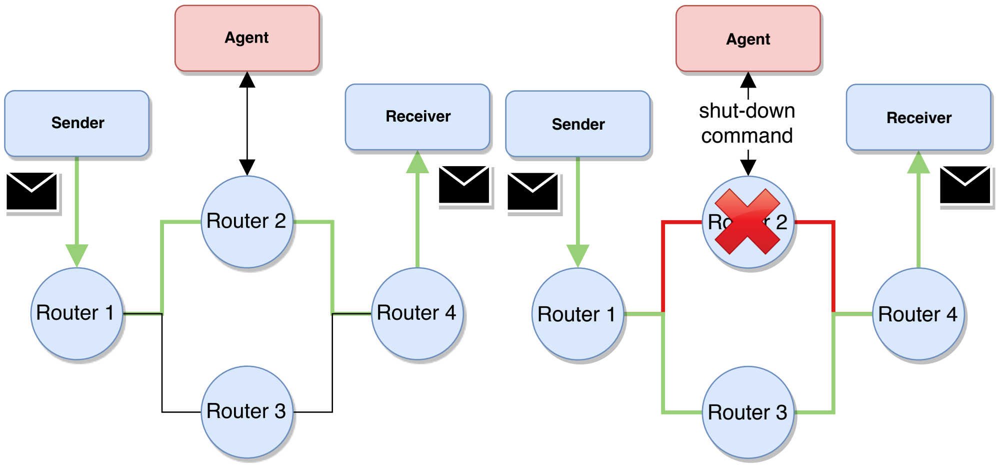


Figure 4.5: The architecture of updated Maestro for testing of the Qpid-dispatch router.

In the Figure 4.6a we show the simple scheme of topology with one agent monitoring the router 2. In this case communication passes through the router 2 and messages are delivered to receiver without problems. The Figure 4.6b 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 then answer the question *How does router shutdown influence the latency between sender and receiver?*



(a) Network with router agent.

(b) Router shut-down demonstration.

Figure 4.6: A simple network with demonstration of router shut-down.

Communication between cluster back-end and user client is realized through Maestro Broker and so for proper message distribution a new topic has to be added. As was mentioned in Section 3.4, Maestro Clients communicate with back-end via specialized commands. Router Agent will accept a new set of specialized commands for router control. This set has to be added to existing Maestro Clients. All additional components or components that required update are highlighted by red color in the Figure 4.5. The example of simple testing topology consisting of two routers and two brokers is also included in the Figure 4.5.

4.4.1 Extension Points

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

4.4.2 Communication with Agent

For the communication inside Maestro testing cluster we use the Maestro Protocol, which was described in the Subsection 3.2. Maestro Clients have to know how to communicate with this new component in the cluster and so it is necessary to add new communication commands. The following lists 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)** — set path to user commands handlers.
- **MAESTRO_NOTE_USER_COMMAND (30)** — execute user's specific command.

4.4.3 AMQP Inspector

The important part of the performance testing are measurements of internal metrics of SUT. Maestro offers Maestro Inspector for this kind of measurements. However, the current version can monitor only Broker, because Inspector is implemented for the specific interface provided by the Broker. Since Broker is written in Java and provides access to JMX[8] via Jolokia[9], we cannot use current implementation of the Inspector for the Qpid-Dispatch 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, while AMQP Management is based on AMQP messages with specific format.

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

- **name** — this property is always set to string property **self**, which refers to itself object.
- **operation** — AMQP management offers classic CRUD operations. For inspect message we will always use the option called **QUERY**.
- **type** — this property represents the 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 the request purpose. The options for request are:
  - *router* — general informations about the router.
  - *router.stats* — detailed informations about the router.
  - *router.link* — informations about route links.
  - *router.node* — general informations about neighbour nodes.
  - *router.address* — informations about addresses on the router.
  - *connector* — informations about connections.
  - *allocator* — informations about memory metrics.
  - *config.autolink* — informations about created auto links.
  - *config.linkRoute* — informations about created link routes.
- **body** — message payload, which is usually an empty list. Exceptions are auto links and link routes requests, which needs additional information in the body.

[8] JMX — http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html

[9] Jolokia — https://jolokia.org/

Collected Data

Data collected by the AMQP Inspector are different than those collected by 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:

- **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 its active connections, addresses, auto links, accepted messages and etc.
  - **Address Count** — number of active addresses at current time.
  - **Connections Count** — number of active connections at current time.
- **Router Links** — informations about all router links which were opened to the router.
  - **Accepted Message Count** — number of accepted messages at current time.
  - **Delivered Message Count** — number of delivered messages at current time.
  - **Released Message Count** — number of released messages at current time.
  - **Undelivered Message Count** — number of 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 then converted to a line chart, which represents collected values for each request. Data collected by senders and receivers remains the same as in the current version of Maestro.

5 Implementation

This chapter describes the actual implementation details of all components, described in the Chapter 4. The main part focuses on the Agent module and AMQP Inspector for Maestro, which we implemented in Java and Groovy languages. The other part describes the Topology Generator — Python package for automatic generation of dispatched topology based on user's metadata. Data collecting and reporting done by Maestro 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 a configuration file for each specific test case. Hence, this initiated implementing of 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 such approach? Since Qpid-dispatch is getting new versions every few months, they can change names of any configuration attributes or even deprecate them. This causes the problem, that when Qpid-dispatch is updated, then the code of Topology Generator has to be reviewed and updated as well, otherwise one risks syntax errors in the configuration files. So 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 create a 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. Input configuration variables are in JSON format, and 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 the 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, or mode, 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
var output = ””
for line in attributes_file :
    if line.is_attribute() :
        output += line.attributeToJinja2()
    else if line.is_section() :
        output += line.sectionToJinja2()
    else
        output += line
output.strip()
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

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

statement. Basically this class is dynamically created for each section and these methods are then invoked before first and after last attribute. The `_enter_()` method wraps start of each section with following code:

```
{%% 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 the installed version of Qpid-dispatch on remote node it guarantees that the template will always correspond with the 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 actor in 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 the 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 (i.e. router or broker) and use them during the 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, generator 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 as well.

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 easily specify for each node informations such as constructors, listeners, SSL profiles, etc.. The whole file is loaded during the initialization and is processed with the Topology Generator.

Topology Type — Topology generator can create topologies without graph file, but then it requires the network type that 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

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

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.

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

Default Connections — default connections correspond 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 the second machine) on each router. These 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 can be SSL or auto-links settings. The generator loads data about these sections from graph file. Qpid-dispatch has a lot of settings, 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 the inventory there has to be a 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 a 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
var inventory = parse_inventory(Inventory)
var graph = create_graph(inventory, Graph File/Topology type)
var output = {}
for node, neighbors in graph.adjacency() :
    output.update(generate_listeners(node, neighbors))
    output.update(generate_addresses(node, neighbors))
    output.update(generate_specific(node, neighbors))
for node, neighbors in graph.adjacency() :
    connectors, link_routes = generate_connectors(node, neighbors)
    output.update(connectors)
    output.update(link_routes)
return output
```

Algorithm 2: Pseudocode of default connectivity generation.

```
Function: generate_connectors()
Input: node—node from graph, neighbors
Output: lists of connectors and link_routes
var connectors = []
var link_routes = []
for neighbor in neighbors :
    if neighbor.is_router() :
        connectors.append(connector_setting)
    else if neighbor.is_broker() :
        connectors.append(connector_setting)
        link_routes.append(link_route_setting)
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 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 to create the Ansible playbook, to run all necessary tools and to deploy 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 features from Ansible role *ansible-qpid-dispatch*[5] such as start and stop handlers. These steps can be added in any playbook or role, and can be used for automatic topology generation and deployment. The necessary inputs are Inventory and topology metadata for each test-case. In the following description you can see the list of all deployment steps, that are executed on the control node (node where we 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 the play one just needs to specify the path to configuration files and Ansible will do all other necessary steps.

3. **Include variables into Ansible** — this step loads the generated variables into the 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 the playbook execution start. After these steps we are ready to execute the last steps on the remote nodes:

4. **Install qdrouter-jinja2** and **generate templates** — qdrouter-jinja2 is used to generate the 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 the successful 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 necessary updates of all other Messaging Performance Tool parts such as commands updates, extension of the Inspector or report changes. The Agent was implemented in Java and Groovy languages.

5.2.1 MPT Preparations

The first step during the development was 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. At first, we need ti implement the main function for the agent, which is built with each new package. After the creation of main we had to create *assembly.xml* which tells Maven which files has to be

[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

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 4.4.1, 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 was 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 handler method based on the received note type. Usually, the `handle()` methods in `MaestroWorkerManager` class only logs actions. For another functionality we have overridden the specific implementations of each peer.

Every action handler script is written in Groovy, and so Maestro needed a Groovy script executioner. For this purpose, we created the class `GroovyHandler`. This class basically checks the handler file whether it is executable and then tries to execute it. The handler file location is specified by the note payload and there 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. So 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 for every external point handler creates its own thread and naturally, the agent must serve other handlers during this time, and not wait 5 minutes for one of them to finish and then handle the others.

```
Function: callbacksWrapper()
Input: externalPointPath, codeDir, note
Output: sendReplyOk() or sendReplyFail()
var thread = Thread()
try
    var groovyHandler = GroovyHandler()
    groovyHandler.setInitialPath(externalPointPath)
    groovyHandler.setWorkerOptions(getWorkerOptions())
    groovyHandler.setMaestroNote(note)
    thread.start(groovyHandler.runCallbacks())
catch
    sendReplyFail()
sendReplyOk()
```

Algorithm 4: Basic functionality of `callbacksWrapper()` method. This method create new thread for each extension point and tries to execute it.

In new threads we execute `runCallbacks()` method, which load all files from extension point directory and tries to execute them. This method is in a specific class, which contains parameters for each execution. The parameters are originally contained in the note's payload. The Algorithm 5 captures `runCallbacks()` method.

```
Function: runCallbacks()
Input: groovyHandler as this class
Output: sendReplyOk() or sendReplyFail()
for file in extensionPointDirectory :
    try
        var grovyObject = loadFileAsGroovyObject(file)
        groovyObject.invokeMethod("setMaestroNote", this.maestroNote)
        groovyObject.invokeMethod("setWorkerOptions", this.workerOptions)
        groovyObject.invokeMethod("setMaestroClient", this.client)
        groovyObject.invokeMethod("handle", this.context)
    catch
        sendReplyFail()
sendReplyOk()
```

Algorithm 5: The method `runCallbacks()` loops over each file in the extension point directory, tries to load each file and executes it.

The other important method of Agent is the override `handle()` 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 features 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 third party software on the node during the test, which can simulate any kind of the unexpected behavior.

The agent is a 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 the front-end through the 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.

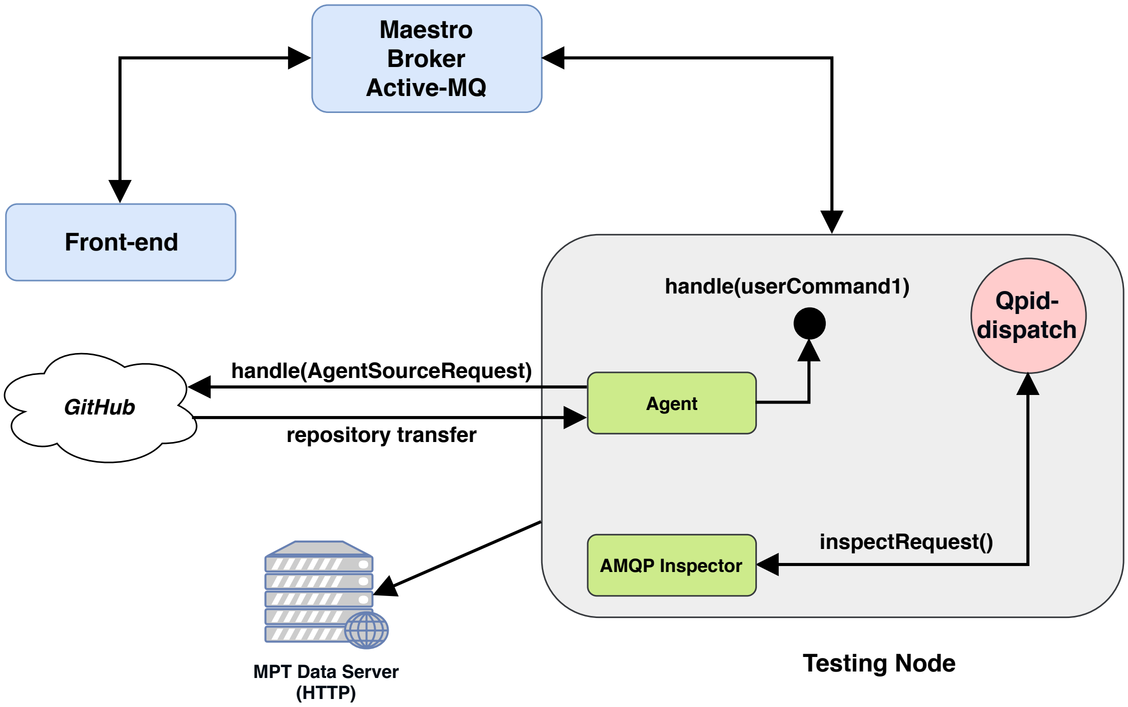


Figure 5.1: Communication scheme inside the Maestro with the agent. Scheme shows the agent git repository download and then handling the proper note defined by the user. The Figure also shown the SUT communication with the AMQP Inspector.

5.2.3 AMQP Management Inspector

The collection of information about the router itself is 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 already 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. While 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 a specific Inspector class based on user definition in the testing script. Currently, Maestro offers ActiveMQ Inspector for the Broker and AMQP Inspector for the Router. The Inspector will receive the note with *inspector start* command, which carries string payload. This payload is the name of the specific inspector implementation that will be started. The mechanism of starting the AMQP Inspector is depicted in the Figure 5.2 and in the Algorithms 6.

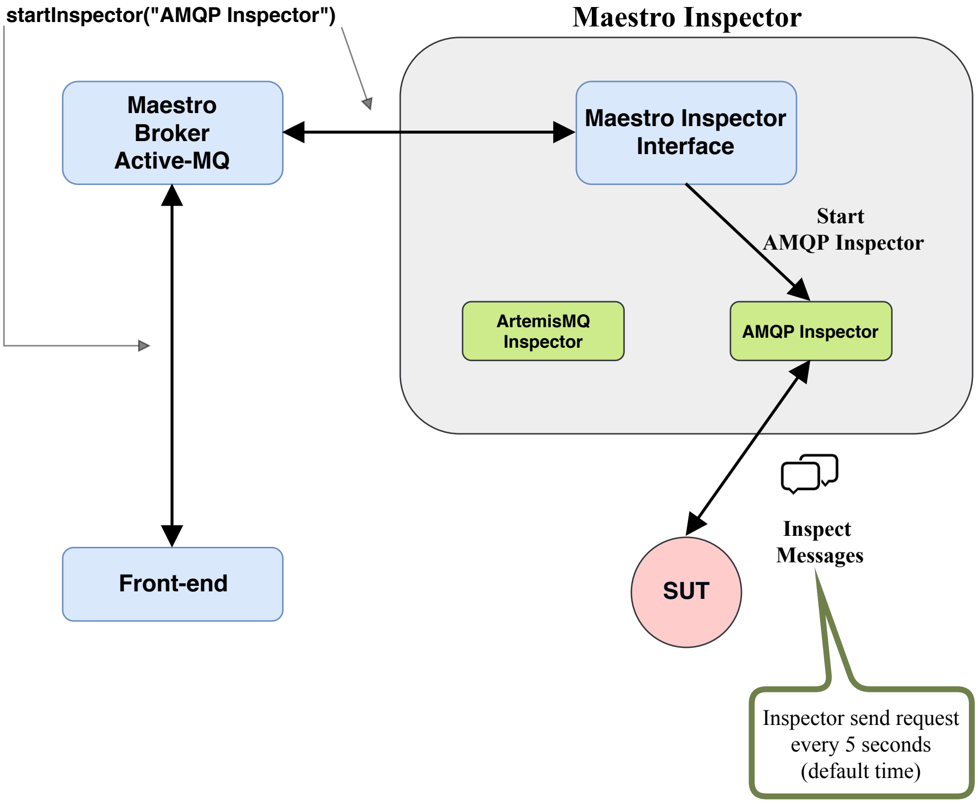


Figure 5.2: The inner mechanism of Maestro Inspector during the receive start inspector note. One can see the note exchange and choose of specific inspector class based on the note's payload.

```
Function: handle()
Input: Maestro note — startInspector
Output: sendReplyOk() or sendReplyFail()
var inspectorClass = note.getPayload()
try
    var inspector = Inspector(inspectorClass)
    var thread = Thread(inspector)
    thread.start() sendReplyOk()
catch
    sendReplyFail()
```

Algorithm 6: Handler method for startInspector note which creates instance of specific inspector implementation.

The AMQP Inspector uses the request-response message mechanism with the SUT. The inspector creates message using Java library *Qpid JMS*[6] as specified in the Subsection 4.4.3. Since we want to collect as much relevant data as possible, we are sending three[7] request-response messages with different entityType option every 5 seconds during the whole test. For the response collecting it is necessary to create a temporary queue, that is used by the router as response destination. The destination is contained in the field *response-to*. The actual request message is represented as an object with type of *JMS Message*. The main Inspector's process mechanism is described in the Algorithm 7, while the message request-response mechanism is depicted in the Figure 5.3.

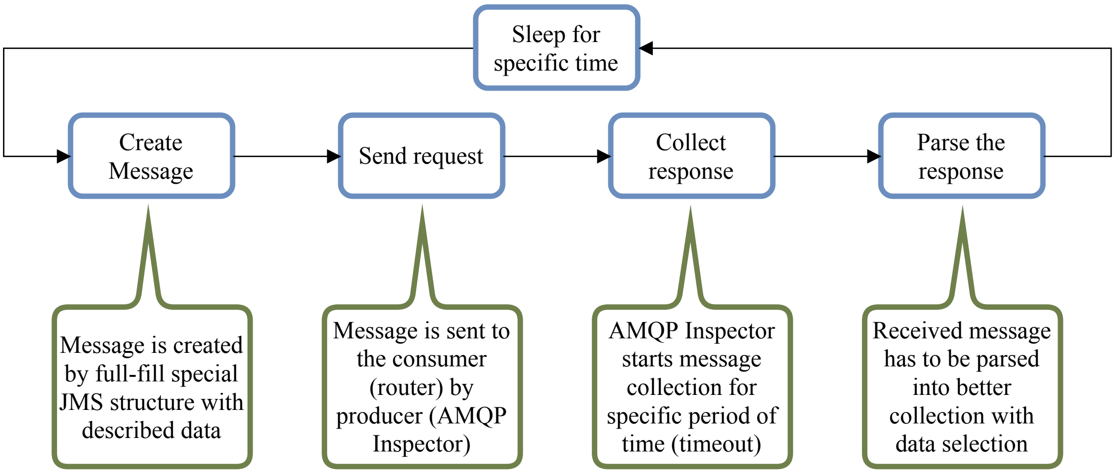

Sleep for specific time
Create Message
Send request
Collect response
Parse the response
Message is created by full-fill special JMS structure with described data
Message is sent to the consumer (router) by producer (AMQP Inspector)
AMQP Inspector starts message collection for specific period of time (timeout)
Received message has to be parsed into better collection with data selection

Figure 5.3: The whole Inspector process including message creation, message sending, collecting and parse.

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

[7] Three specific requests to AMQP Management are enough to collect all data which are needed.

```
Function: start()
Output: sendReplyOk() or sendReplyFail()
var routerLinkInforWriter = RouteLinkInfoWriter()
var memoryInfoWriter = MemoryInfoWriter()
var generalInfoWriter = GeneralInfoWriter()
try
    var currentTime = System.currentTimeMillis()
    connectToRouter()
    var dataReader = DataReader()
    while canContinue() do
        routerLinkInforWriter.write(currentTime, dataReader.collectRouterInfo())
        memoryInfoWriter.write(currentTime, dataReader.collectMemoryInfo())
        generalInfoWriter.write(currentTime, dataReader.collectGeneralInfo())
        Thread.sleep(5000)
    end
    sendReplyOk()
catch
    sendReplyFail()
```

Algorithm 7: Method for starting new instance of the Inspector. This method continuously sends requests to the SUT, collects, parse and write the response into csv file.

6 Experimental Evaluation

This chapter summarizes results of the performance testing and experimental evaluation of Maestro. We split the experiments into two parts. The first performs a basic measurement of Maestro 1.3.0 which includes Maestro Agent and AMQP Inspector. During this experiments we focused on reclaiming the highest possible throughput of singlepoint topology of Qpid-dispatch and Message Broker and multipoint topologies with three nodes of Qpid-dispatch and with Broker in the middle. These experimental topologies are depicted in the Figure 6.1. The later series of experiments is focused on behavior testing of topologies, which involves Qpid-Dispatch reliability and recovery testing.

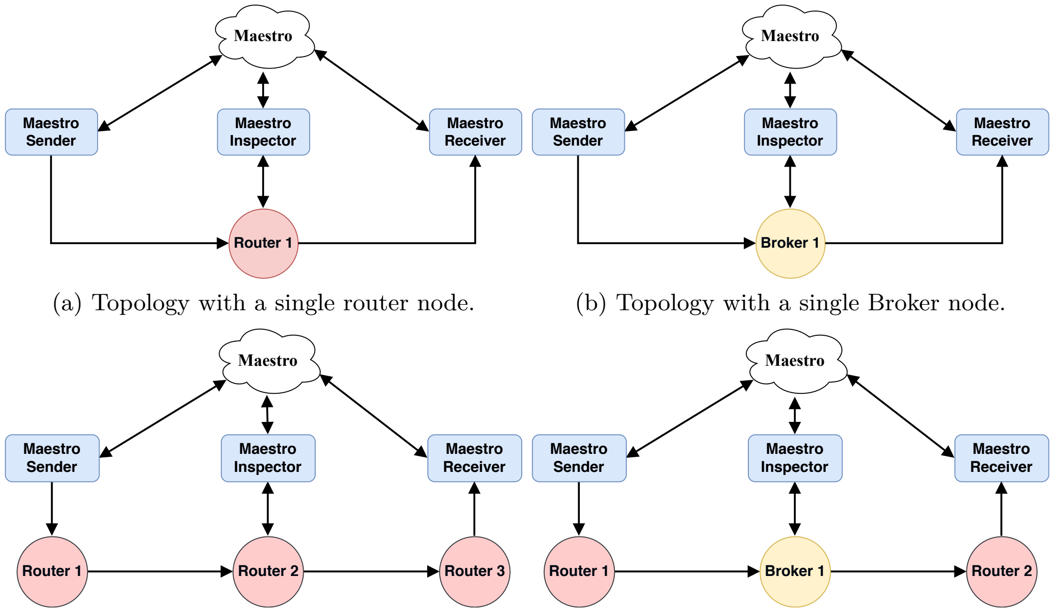
Since the testing was executed over multiple topology types, we used Topology Generator for quick automatic changes of topology. Each test was executed against established topology where all components were newly installed and restarted between each test scenario. This was done during the cleaning stage. For experimental evaluation we used machines specified in the Table 6.1. The reason why clients use more powerful machines is that we needed more machines for SUT, but only two IBM Xeon machines were available during the experimental evaluation and we needed at least three machines for the SUT nodes. For proper comparison we need all SUTs on the same machine type.

Table 6.1: Machines and their properties, which were used for the experimental evaluation.

	Machine	**CPU**	**RAM (Gb)**
SUT	Opteron	8	8
Clients	IBM Xeon	16	16

6.1 Basic Performance Measurements

Maestro works as the orchestration system, and requires proper infrastructure before one can 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 obviously rises with more complex scenarios and larger networks. Examples of used generated experimental topologies are depicted in the Figures 6.1. For these configurations we compared the throughput and latency of these combinations. During all measurements we used Maestro Inspector to inspect one of the SUT depending on the topology type. Note, that for Qpid-Dispatch we used AMQP Inspector and for Broker we used ActiveMQ Inspector. The topologies were picked based on current performance testing and known topologies, where some performance degradation was already found during the previous testing.



(a) Topology with a single router node.

(b) Topology with a single Broker node.

(c) Topology consisting of routers nodes only.

(d) Topology with Broker in the middle.

Figure 6.1: Examples of experimental topologies created for basic performance testing and experiments with Maestro. The arrows indicates the communication path between topology components.

6.1.1 Throughput

We measured throughput only by load generators — *Maestro Sender* and *Maestro Receiver*. Load generation depends on the test properties as one can see the test properties for each test case in the Table 6.2. 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 Message Broker. The unbound rate during the test is achieved by setting the environment variable *RATE* to value 0. The throughput test cases are focused on maximum throughput of simple or complex topologies.

Table 6.2: Test case settings for throughput measurements

Test Property	**Value**			
	Singlepoint		**Multipoint**	
	Router	**Broker**	**Full Router**	**With Broker**
MESSAGE_SIZE [B]	256			
PARALLEL_COUNT	5			
TEST_DURATION [m]	15m			
RATE	0			

Single Node

The first tests were ran against the single node topologies, which are depicted in the Figures 6.1a and 6.1b. These topologies contains only one SUT node, which is forwarding messages from sender to receiver. During the test the SUT node is inspected by the proper Maestro Inspector.

The measured throughput is depicted in the Figure 6.2 where one can see the comparison of tests with 15 minutes duration, which tries to achieve the highest possible throughput. One can see that the maximum throughput of Qpid-Dispatch, as a standalone network component, can reach around 90 000 messages per second. On the other hand, the lone Messaging Broker reaches only about 30 000 messages per second. This throughput difference is caused by the fact, that Broker stores all of the messages in the memory until clients want them. This is the main feature of the broker, because it operates as an message distributor in the network. The router only routes the messages to the destination which does not need to store message in the memory.

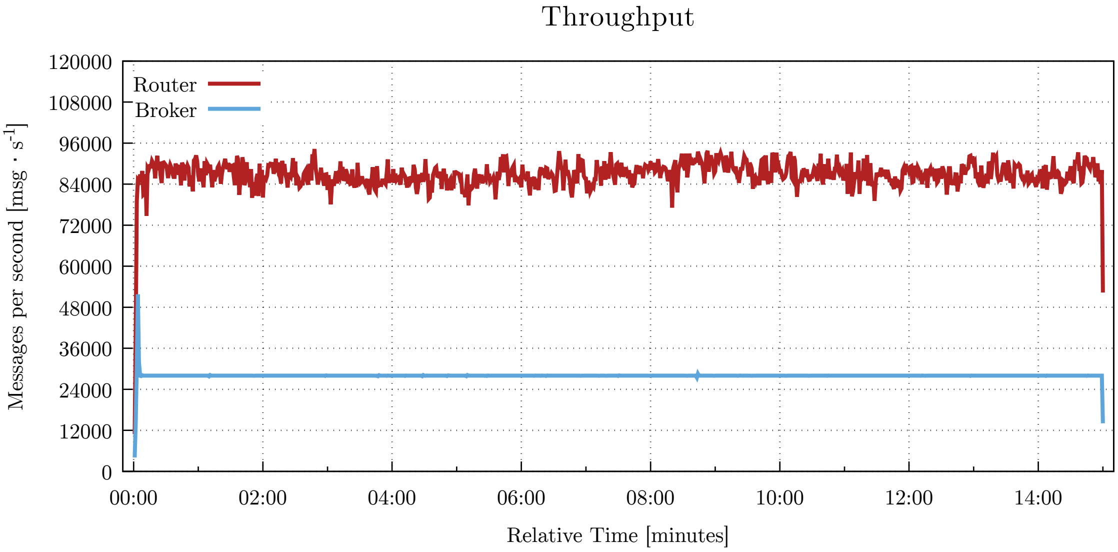


Figure 6.2: Chart of the maximum throughput of router and broker during the singlepoint test case. One can see the significant difference between those two components.

In the Figure 6.3 we can see the memory usage of Qpid-dispatch during the test. We can see here, that the totally allocated memory is around 45 kB from which it is used only around 13-28 kB. If we compare this with the memory allocation for the Broker, we can see the huge difference between these values. The memory allocated for the Broker is depicted in the Figure 6.4 and we can see that the allocated memory is around 2 GB of memory and used memory is around 300-900 MB. This is caused by messages stored in the memory.

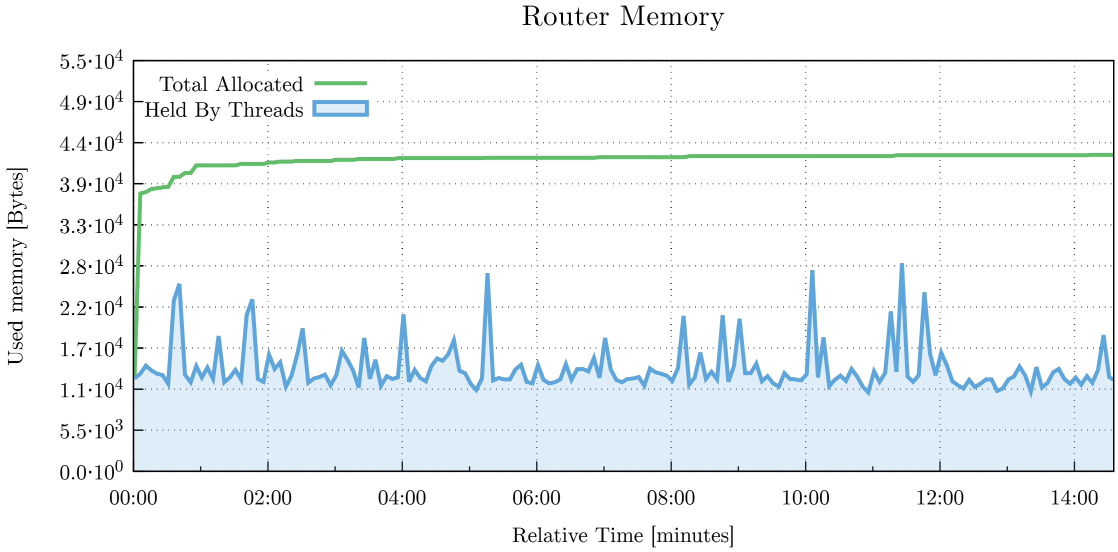


Figure 6.3: The total allocated memory and memory-in-use by Qpid-Dispatch during the test. The data were collected by the inspector every 5 seconds.

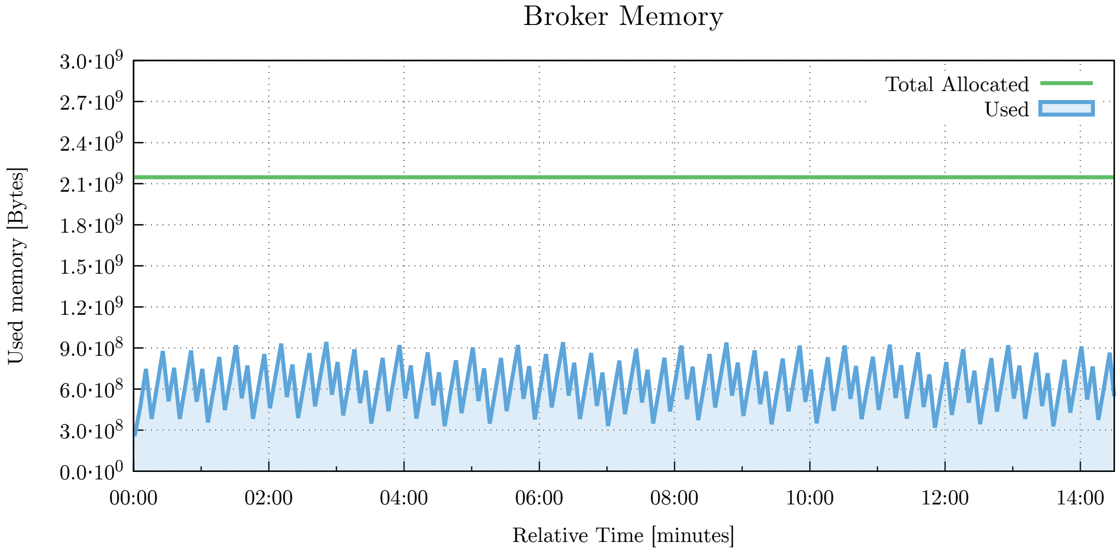


Figure 6.4: The total memory allocation for the Broker service. One can see that the broker allocates more memory compared to Qpid-Dispatch in the Figure 6.3.

Multipoint Topology

For the multipoint experiments we used topologies depicted in the Figures 6.1c and 6.1d. The network throughput can be influenced by other device connected to the topology. The singlepoint topology was extended by another components by adding two other routers around the original SUT. The versions of the newly added SUTs are the same as the original ones.

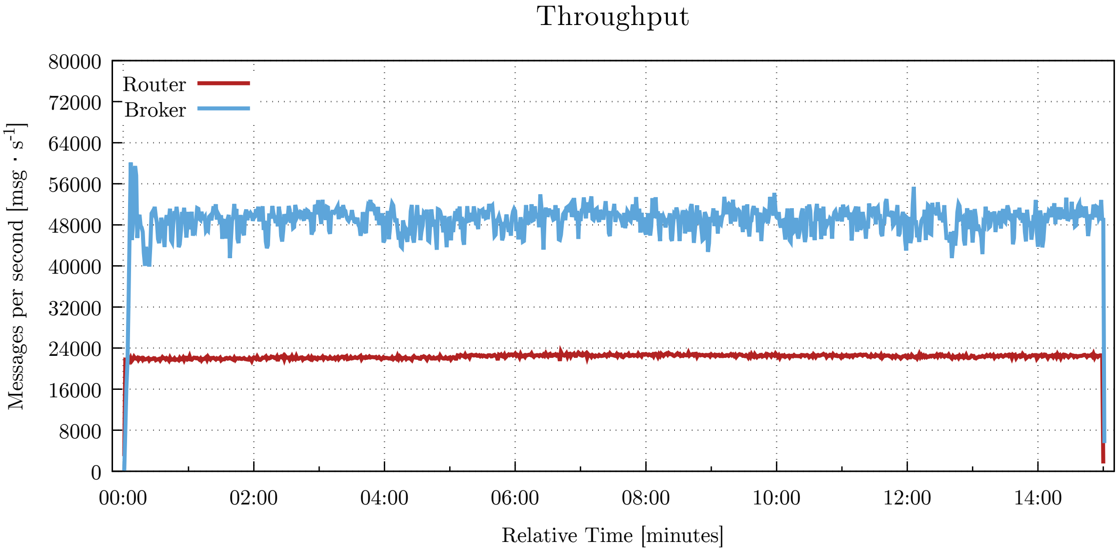


Figure 6.5: Measured throughput of Qpid-Dispatch and Message Broker during the multipoint case study. One can see the performance degradation of Qpid-Dispatch and improvements of Message Broker on that Figure.

In the Figure 6.5 one can see, that adding routers to the broker node raises achievable throughput to the 48 000 messages per second. On the other hand, the topology consisting only of the routers has significant performance degradation. The throughput falls from the 90 000 messages per second to the approximately 23 000 messages per second. This degradation is caused by the interior flow-control mechanism, which should prevent the overload of the network. However, in this case study we can see that the performance degradation is too high and the mechanism used in the Qpid-Dispatch should be re-implemented.

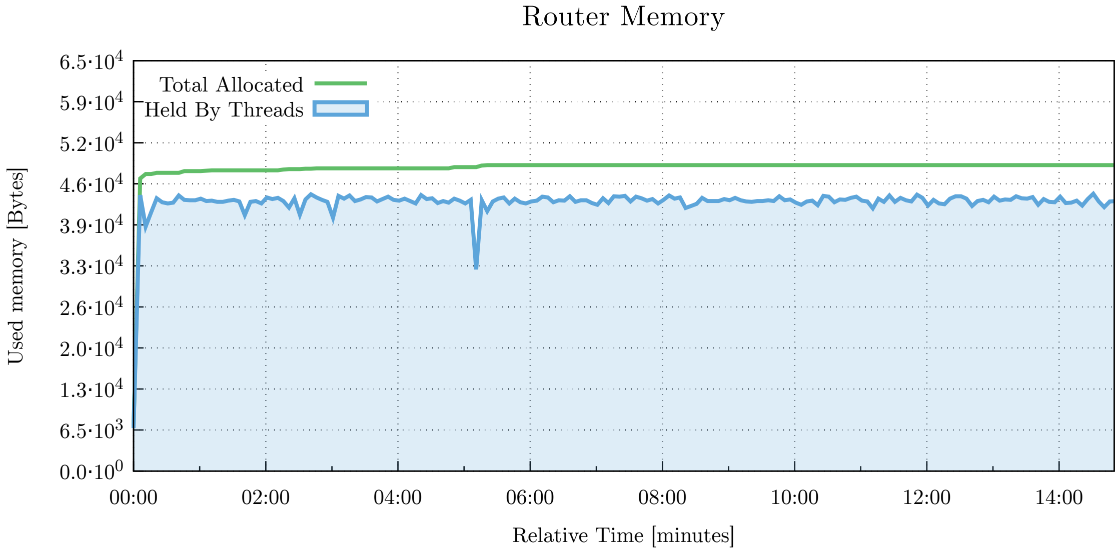


Figure 6.6: Qpid-Dispatch's memory usage during the multipoint case study. Used memory is higher than in the single-point.

Based on that mechanism, the memory usage of the middle router depicted in the Figure 6.6 is higher than during the previous case study. Memory used by all threads is around two times higher and the mean value is around 43 kB. On the other hand, the memory allocated for the broker component remains the same as in the previous case study. The memory monitoring for this case study is depicted in the Figure 6.7.

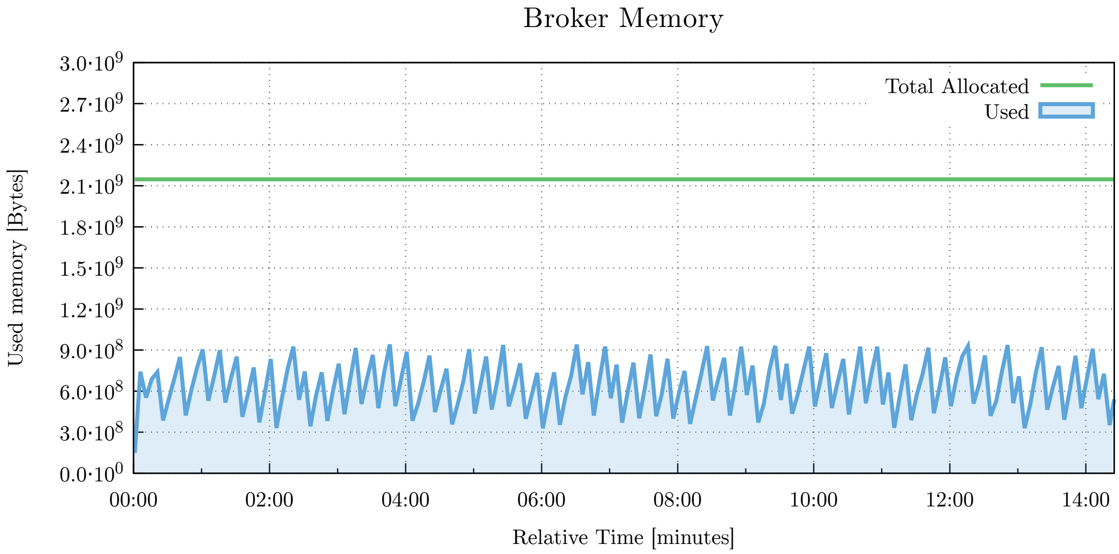


Figure 6.7: Memory usage for Broker remains almost the same as in the single-point case, but with less spikes.

Conclusion

The collected data during the throughput measurements reveal unexpected and considerable performance degradation in the serial connection of the Qpid-Dispatch. The comparison between the single and multi-point case study is in the Figure 6.8, which groups together all throughput measurements data into one chart. Here one can see the performance improvement between single instance Broker test and the test of topology with the broker (yellow and green color), and performance degradation between router topologies (red and blue color). The summary of results is also available in the Table 6.3.

Table 6.3: Table with collected data with highlighted performance improvements and degradations.

Test Type	**Throughput (Msg/s)**		**Memory**	
	Expected	**Measured**	**Total**	**Used max**
Single Router	-	90 000	45 MB	28 MB
Single Broker	-	30 000	2 GB	0.9 GB
Line of Routers	90 000	23 000	49 MB	43 MB
Line with Broker	30 000	48 000	2 GB	0.9 GB

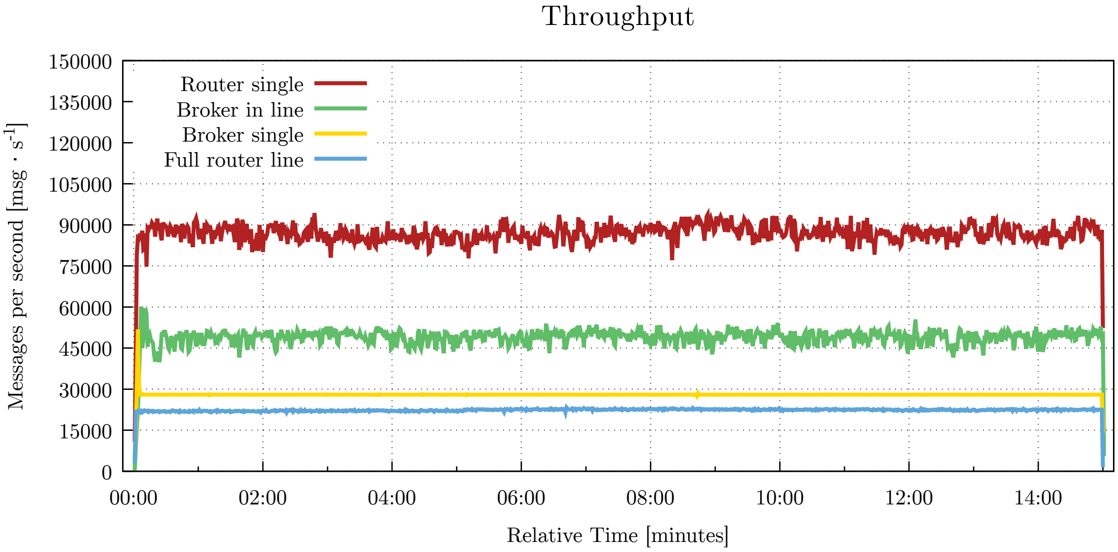


Figure 6.8: The comparison of all measured throughputs for different components and topologies.

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 if one wants better latency during the communication, especially in the case of topology with the 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. The test cases for latency measurements has slightly different settings than throughput measurement. The settings for this measurements are shown in the Table 6.4. Note, that *RATE* and *TEST_DURATION* are sets for each of five connected clients, which means that test is finished after sending 10 000 000 messages.

Table 6.4: Test case settings for latency measurements.

Test Property	**Value**			
	Singlepoint		**Multipoint**	
	Router	**Broker**	**Full Router**	**With Broker**
MESSAGE_SIZE [B]	256			
PARALLEL_COUNT	5			
TEST_DURATION	2 000 000 messages			
RATE	15 000	4 600	3 600	7 600

Single Node

The latency measurements are done with 80% of maximum rate, which were discussed in the Subsection 6.1.1. In the Figure 6.9 you can see the latency difference that we measured between Qpid-Dispatch and Message Broker. In single node measurements, the router's latency is slightly higher in the most of the cases. This is caused by the Maestro Sender and Receiver technologies, because they are implemented in Java and Maestro Sender and Receiver are using JMS for sending and receiving messages, which is same approach as the Broker uses. For better latency comparison it is necessary to measure Broker's latency with current implementations and Qpid-Dispatch's latency with Python clients. However, the current version of Maestro offers only the JMS clients.

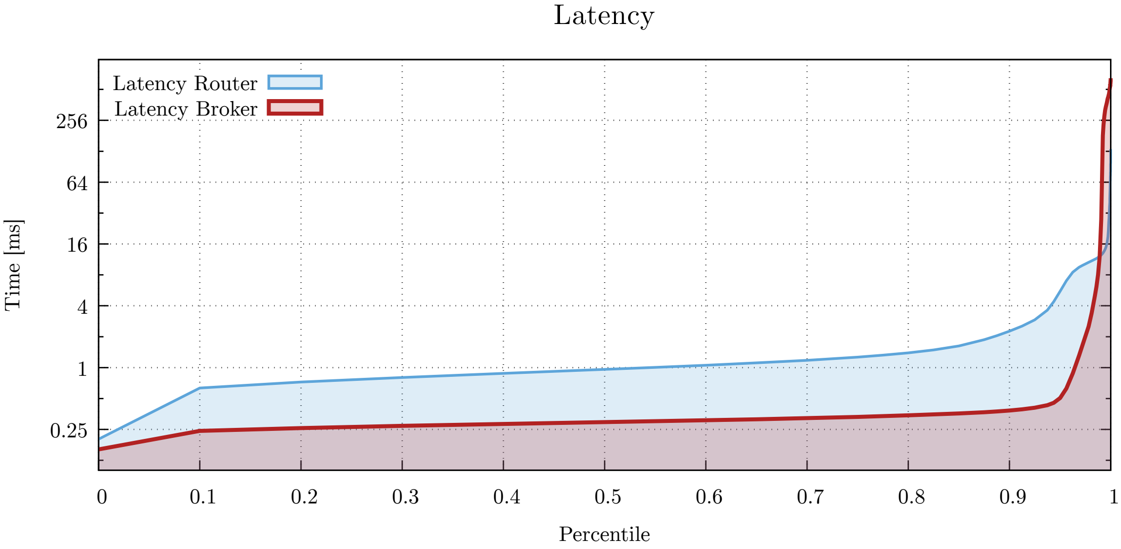


Figure 6.9: Latency chart showing the difference between the router and the broker latency at 80 % of maximum rate. One can see that router is slower, but this is caused by the technology used in the Maestro Clients which is same as in the Broker.

The memory used by Qpid-dispatch is slightly lower and much stable than in the case of maximum throughput as one can see in the Figure 6.10. This proves, that used memory is dependent to the load. If the load on the router is higher then it needs more memory for proper routing. In the Figure 6.11 one can see the Inspector output for Broker's used memory. The used memory here is much stable than in the previous cases, which is caused, as in the router case, by lower load on the Broker. Maximum used memory stags the same as in the previous cases.

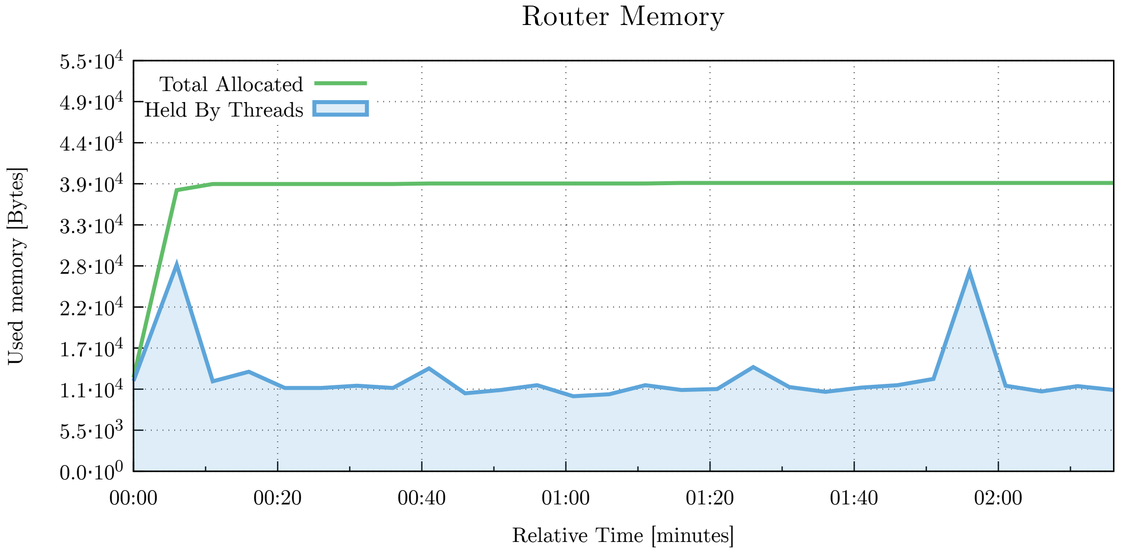


Figure 6.10: Memory usage of Qpid-Dispatch is much stable when the router is not under the maximum load. The spikes are caused by some unexpected events in the topology.

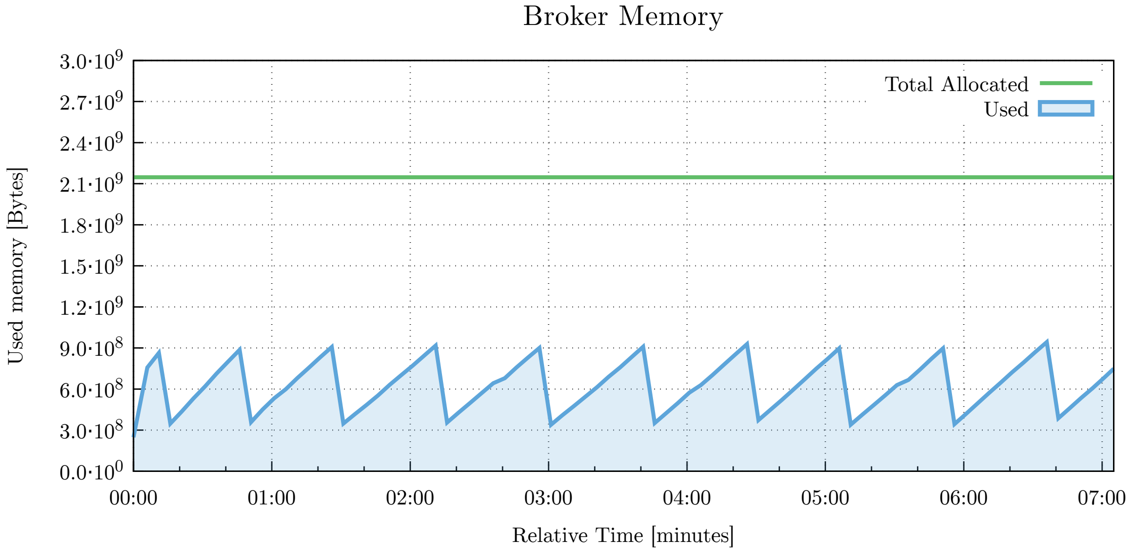


Figure 6.11: The Broker's memory usage has lesser spike when the load is only about of 80 % of maximum.

Multipoint Topology

One can see the measured latency on multinode topology of three routers, and two routers with middle-broker in the Figure 6.12. The latency curve proves, that routers are able to deliver messages into its destination faster than the topology with the Broker, again because the Broker needs to store them in the memory. The latency of the topology with broker reaches around 16 μs in 90 % of samples; on the other hand, topology consisting of routers has significantly better latency that is around 1 μs in 90 % of samples. The conclusion is that the collected data proves the router should be much faster than the broker during the certain circumstances..

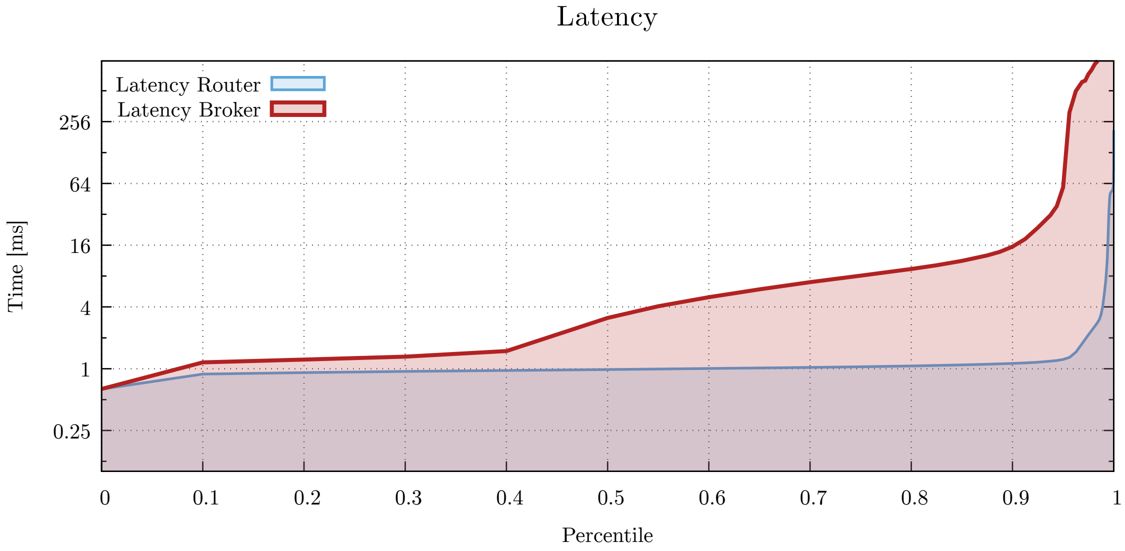


Figure 6.12: Latency comparison between topologies with only routers and with the middle-broker. The router network is here significantly faster.

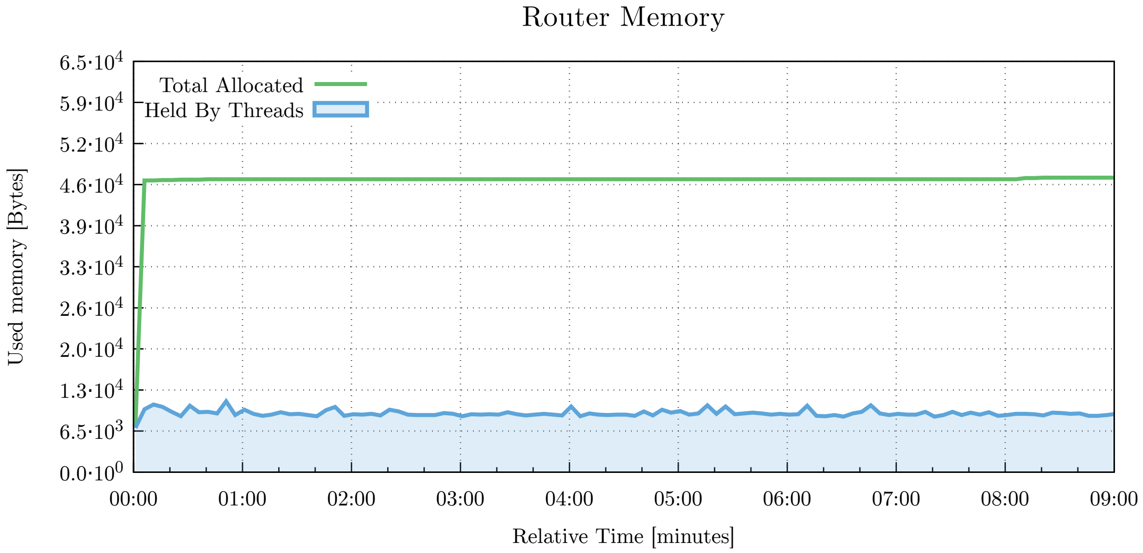


Figure 6.13: Memory usage shows, that memory usage of the router is affected by the throughput.

Collected data about the memory usage proves the previous statements. In the Figure 6.13 we show used memory by Qpid-Dispatch. The curve is very stable and the values moves around the 9 MB of used memory. The used memory by the Broker is shown in the Figure 6.14 and is very similar as in the previous measurements.

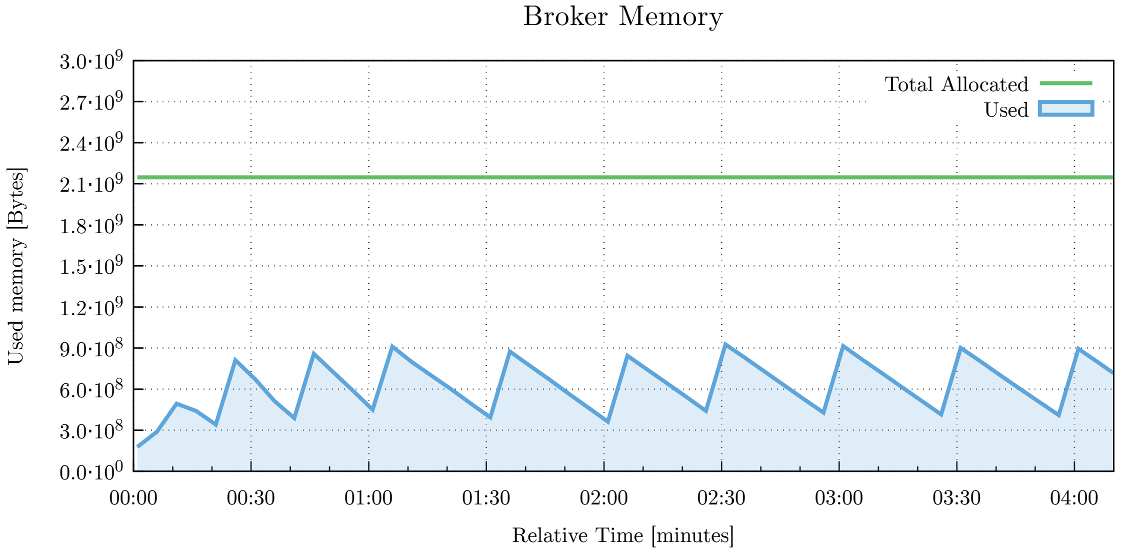


Figure 6.14: Chart of memory allocation on the Broker node..

Conclusion

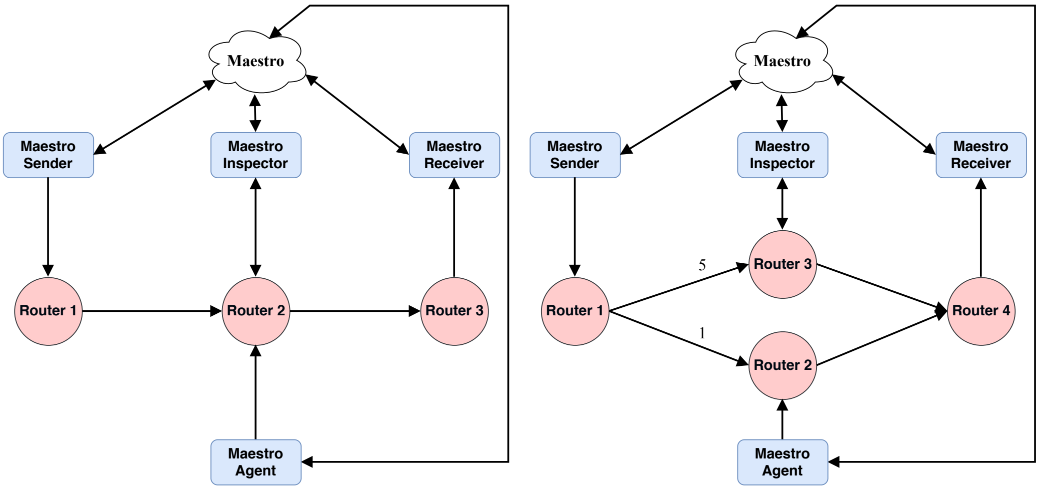
During the latency measurements we collected and compared data for the Qpid-Dispatch and Message Broker topologies. The summary of latency measurements is available in the Table 6.5. Is it was already mentioned, Qpid-Dispatch is faster in the model environment the Message Broker. How to achieve better comparison of latency measurements is discussed in the Section 7.6 as a part of possible improvements.

Table 6.5: The summary table with collected latency data with highlighted performance improvements and degradations.

Test Type	**Latency (us)**		**Memory**		**Duration**
	90 %	**99 %**	**Total**	**Used max**	**(seconds)**
Single Router	2.263	12.495	38 KB	28 KB	136
Single Broker	0.386	181.759	2 GB	0.9 GB	425
Line of Routers	1.292	50.815	46 KB	8 KB	540
Line with Broker	15.487	1031.167	2 GB	0.9 GB	250

6.2 Behavior Measurements

Moreover, we present some results collected during the behavioral testing with using the Maestro Agent extension. The topologies used in the following scenarios are depicted in the Figure 6.15. The topology depicted in the Figure 6.15a is used for demonstrate Agent functions and message loss during the crash. The other topology depicted in the Figure 6.15b represent basic line link with redundant router 2 which is configured as a slave and root router 3 which is master. Here we demonstrate recovery time of Qpid-Dispatch.



(a) Topology with a single Broker node.

(b) Topology with a single router node.

Figure 6.15: Examples of experimental topologies created for behavioral performance testing and experiments with Maestro. The arrows indicates the communication path between the topology components and the numbers represent cost for the path.

On each topology was performed four tests with different actions performed by the Agent. The test properties remains the same as during the latency testing for router line topology with the difference in test duration, which was set to six minutes. The following actions, with additional parameter such as duration, were performed during test:

- **Restart** — simple router restart.
- **Shutdown** — simple shutdown and start for different time duration.

6.2.1 Agent Demonstration

The agent performed specific action in the third minute of test scenario (there can be small delay caused by repository download on the Agent). The shutdown actions has specific duration, which was set to 10, 60 and 120 seconds. Since topology used for this type of tests does not have any redundant path to destination or Broker work message store, the messages got lost during the actions. Note, that the test was triggered without message acknowledgment settings for the router and the clients. In the Figure 6.16 one can see the throughput affected by the restart and shutdown actions in every case study. The magnitude of the action impact is based on the action duration, hence the longer shutdown will provide more message lost than short restart. However, the chart proves, that routers can establish lost connection with the clients without problems when the router is started.

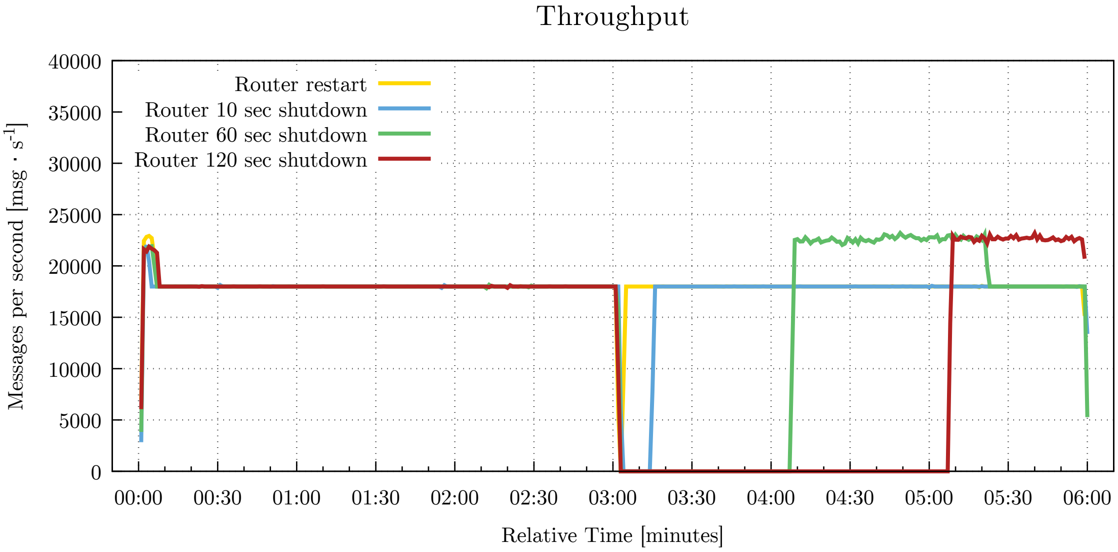


Figure 6.16: Maestro Agent demonstration against a simple topology with actions restart and shutdown in the third minute of test.

The latency is affected as well, one can see that significant message amount raise the latency from 1 μs to 64 μs. However note, that some messages was lost which lead to smaller number of samples for latency compute. The message lost ratio is captured in the Table 6.6.

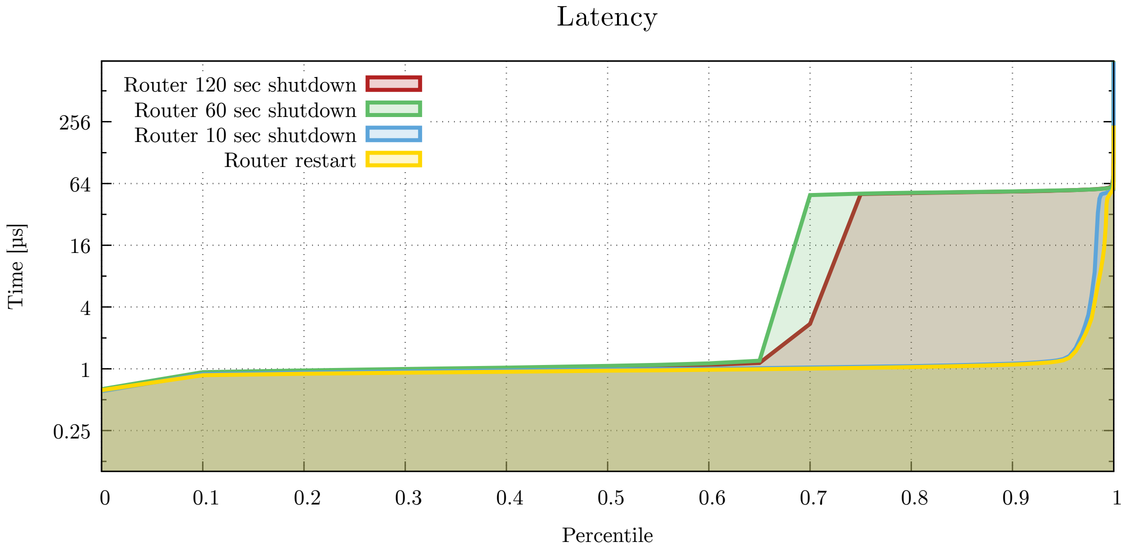


Figure 6.17: Latency diagram affected by the actions performing the connection issues.

Table 6.6: Table with summary of lost messages during the perform specific action on the router node. **[[upravit hodnoty podle new testu]]**

Action	**Duration (s)**	**Message Lost**		
		Expected	**Lost**	**Percent**
Restart	0	7 500 000	XXX	X %
Shutdown	10		XXX	X %
	60		XXX	X %
	120		XXX	X %

6.2.2 Measurement With Redundant Router

During this experiment the Agent perform the same actions as in the previous test cases. The difference is, that current topology has slave router connected into the network which is ready to route the messages when master router crash. In the Figure 6.18 is depicted the throughput for all tests on this topology. The Agent performs actions in third minute which caused spike under the stable load curve, but the throughput is raised back quickly. This spike is caused by small delay when the redundant router starts his job. It needs some time for warm-up, which involves the memory allocation depicted in the Figure 6.19. As one can see, there is no additional spikes after then master is turned up, hence the first spike is cause only by first routing redundant router.

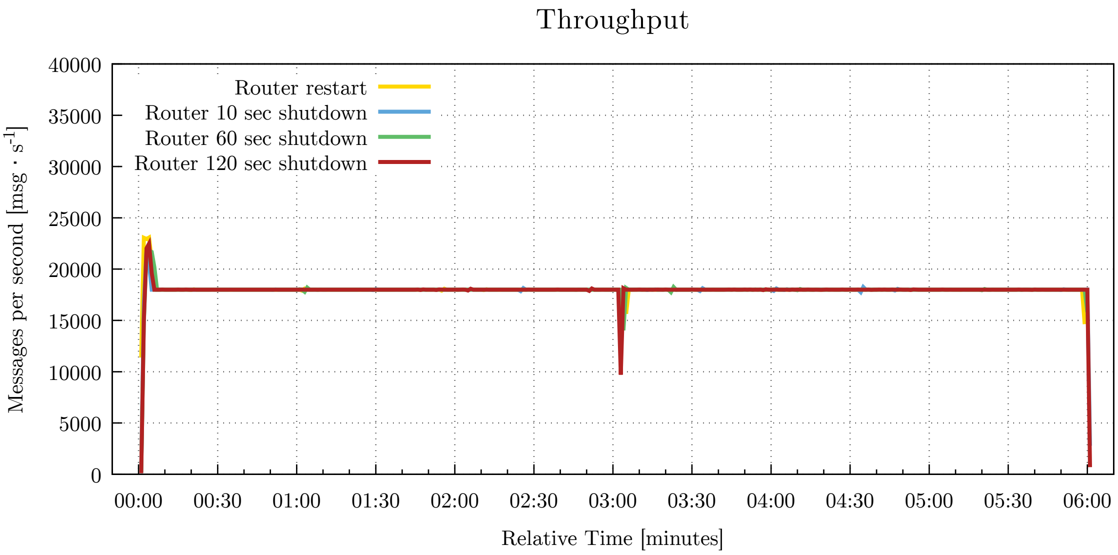


Figure 6.18: Throughput comparison between the test cases with different Agent executions. The spike is caused by warm-up period of redundant router.

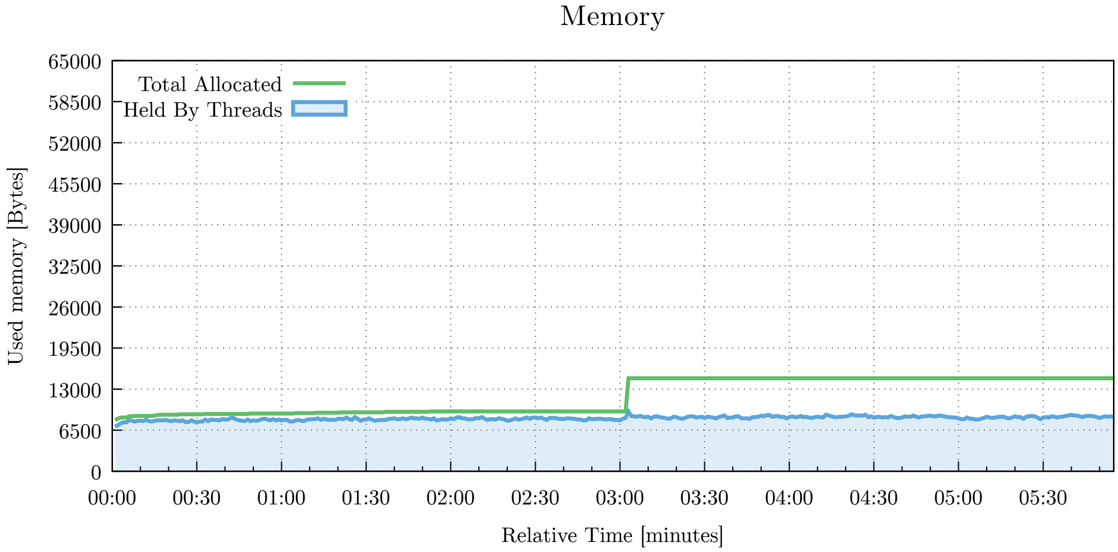


Figure 6.19: Allocated memory for redundant router during the restart. One can see that router allocate new memory when the master router crash and the slave has to handle the load. This memory is allocated till the tear down.

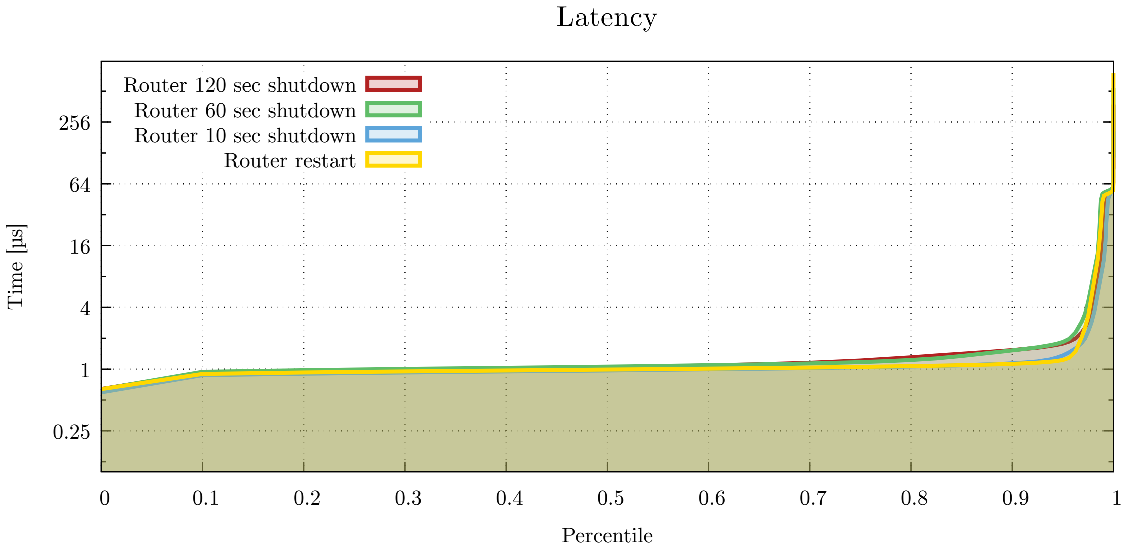


Figure 6.20: Latency diagram with Agent actions performing and redundant router in the topology. The latency remains the same for all the test cases which points to good routing between the routers.

Since we want to know how long it takes to router to re-establish connections after crash we can find the answer in the Figure 6.21. One can see the detail of test case with restart router action which is executed three minutes after the test starts. The monitored router is the redundant one, so we can see that it handled load for two seconds. After this time the master router was able to route load again and the slave router just waits for another communicaiton.

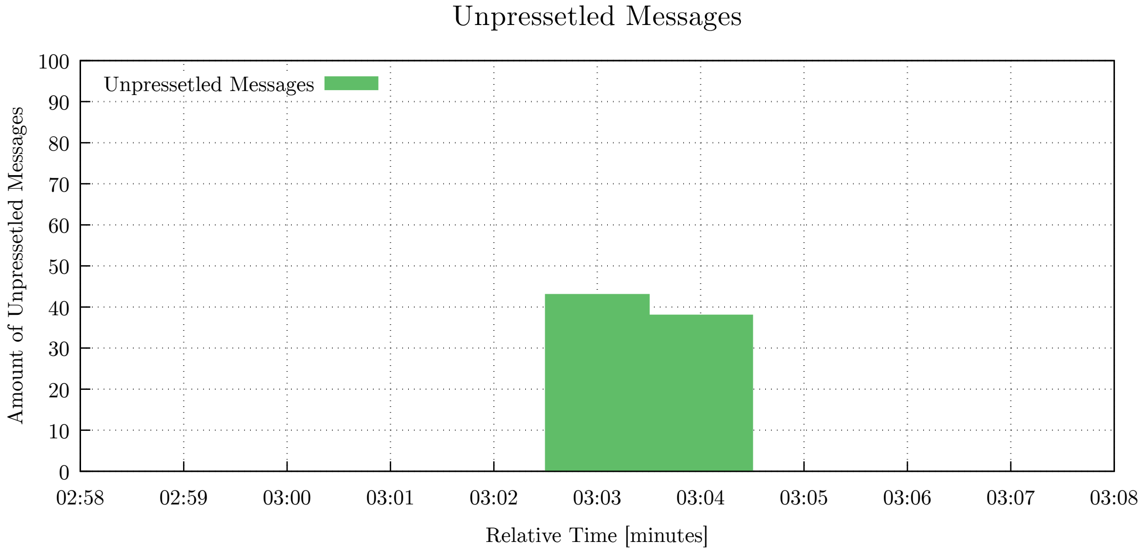


Figure 6.21: Chart captures unpressetled messages on the redundant router node. The slave router handled load for two seconds.

The conclusion is, that Qpid-Dispatch is able to recover after crash in less than three seconds, when there is no block for service start. When the router is down, the topology is updated and the previous hop does not have path to the crashed router, so the clients cannot affect the router start after the crash.

6.3 Problems

[[**Todo**]] All measurements were orchestrated by an automation server called Jenkins[1].

[1] Jenkins — `https://jenkins.io/`

7 Future works and ideas

The Maestro is currently used for performance testing of Message Broker and Qpid-dispatch in Red Hat Messaging team. This makes the Maestro one of the key utilities for the Messaging and the primary performance tool. Since the Maestro is basically immature system, there is still a lot of places for improvements. We present several ideas in the following. Note, that Maestro-Agent and AMQP Inspector are new Maestro modules, which makes performance testing of Qpid-dispatch with collecting interior data about SUT itself available. This extensions was already merged in upstream and are available since Maestro stable version 1.3.0.

7.1 Regression Testing

Since both Message Broker and Qpid-dispatch have new builds every few weeks, there can occur a performance degradation. This issue can be caused by just one simple commit, which can fix some issues but break performance. However, Maestro can catch such performance degradation early in process, if there is already previously measured data with specific informations (so called baselines). Maestro can then re-run the same test with new version of SUT and compare the collected results with previous data set.

This mechanism is simple to achieve. The first step is to configure the pipeline job on orchestration and integration system such as Jenkins or Travis CI. This job has to have access to SUT repository and baseline data tagged as a performance standard for the SUT. The trigger of this pipeline can be every push or every commit with specific tag. The other step is the extension of the Maestro-Reporter, where it can compare older data with newly collected ones and report, how much they differ and where. This pipeline job then can alert engineers, that some specific commit caused performance degradation and also show the difference between actual collected data and estimate collected data.

This type of testing can also be applicable to all test cases with different SUT configuration. The Maestro would be able to compare expected data with collected data and tell us that this specific configuration has a performance degradation.

7.2 Data Reporting

The current reports, created by Maestro itself, contain charts, in the *png* format generated by Java library for creating bitmap figures. This makes them less informative that they could be with better data visualization. Since Inspectors collect additional data about SUT, e.g. memory usage, it will be helpful for engineers of SUT to see interactive charts with collected data. With this options, engineers can better analyze what is going on with SUT during the test scenario.

A good example of interactive and vector charts is *Grafana*[1]. Grafana can produce awesome outputs from collected data e.g. from the database. Another example is *Project Jupyter*[2], which can plot interactive charts from database source data on the fly. One only needs installed Python on the node. Jupyter starts a Python server on the node and makes plotted data available via the HTTP browser. Maestro can implement such strategy, as a new peer similar to the data server code, which is running on all Maestro peer nodes. The difference is, that this report server will be started by Maestro-Reporter on the execution node.

7.3 Collected Data Compression

Each Maestro peer collects different data during the test. Size of these data is based on peer type, collected data format and test duration. For example the Maestro-Receiver collects huge amount of time for throughput and latency chart. These data are represented as a double-column csv file with columns *eta*(estimated time of arrival) and *ata*(actual time of arrival). Each csv file looks like the following:

```
eta;ata
"2017-10-19 13:19:32.661300","2017-10-19 13:19:32.706649"
"2017-10-19 13:19:32.661500","2017-10-19 13:19:32.706823"
```

Imagine, that this record is written for each send/received message on sender or receiver. For example we can have 50 000 records with prefix „`2017-10-19 13:19:32"` which represents a huge redundancy. The idea of compression is to save only first timestamp and then compute difference between saved timestamp and current timestamp and write this difference into csv file. This way would be able to save at least 15 Bytes per timestamp, which saves more than one half of current size. The only thing is to write a new timestamp after some time, when difference is too big. The new csv file would then look like the following:

```
eta;ata
1525285541559,1525285560346
+30,+40
+35,+42
```

7.4 Multi-point Senders and Receiver

Behavioral testing introduces an idea of multipoint senders and receivers. Lets say, that we want to collect behavioral data about Qpid-dispatch with two queues, where the first queue accepts messages from two senders and the second queue accepts messages from five senders. This situation better simulates the real network traffic than the current mechanism. To achieve this, the Maestro needs to extend Maestro-Worker with option for multiple endpoint connections dynamically. The current version offers only one specific connection specified by the user.

[1] Grafana — open source software for time series analytics https://grafana.com/

[2] Jupyter — http://jupyter.org/

7.5 Maestro-Agent Executor Improvements

The Maestro-Agent is able to download external git repositories and tries to process them during the test. However, the external code handler is currently designed only for code written in Groovy. This limitation can be easily removed by creating more general executor, which would be able to execute any type of scripts. One idea how to achieve this is to create more complex executor in *Kotlin* languange[3]. The new executor should be able to run each type of downloaded script and keep the access to the return code and standard output. This extension would remove the limitation to the use, which has to specify each external action handler in the Groovy language. Note, that new executor should not affect performance testing during the execution, so the operations should remain atomic.

7.6 Maestro Clients in Python

In the Subsection was mentioned, that latency of Qpid-Dispatch and Message Broker cannot be compared with lower values. The reason is the used technology for Maestro Sender and Maestro Receiver. Both of them are written in Java (JMS) which makes the connection to the Message Broker (written in Java, using JMS) faster then connection to another technology. This makes Qpid-Dispatch slower in case of connection creation, since it is written in Python and C (using Qpid-Proton library for connections). For more precision during the latency measurements, the Maestro should use Python clients for load generation in case of Qpid-Dispatch testing. However, current version of Maestro offers only JMS clients for performance testing, which makes more space for future improvements in scenarios like Qpid-Dispatch latency testing. Note, that delay caused by connection creation is in a matter of microseconds.

[3] Kotlin — https://kotlinlang.org/

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] Fulay, A.: Containers Deep Dive – LXC vs Docker. Online. [visited 2018/05/16]. Retrieved from: https://robinsystems.com/blog/containers-deep-dive-lxc-vs-docker-comparison

[14] 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

[15] 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/

[16] 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

[17] 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

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

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

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

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

[22] 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.

[23] 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.

[24] 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

List of Figures

List of Tables

List of Shortcuts

MPT	Messaging Performance Tool
KPI	Key Performance Indicators
CPU	Central processing unit
SUT	System Under Test
AMQP	Advanced Message Queuing Protocol
MOM	Message-oriented middleware
STOMP	Streaming Text Oriented Messaging Protocol
OpenWire	Cross language wire protocol
REST	Representational State Transfer
JVM	Java Virtual Machine
JMX	Java Management Extensions
SASL	Simple Authentication and Security Layer
SSL/TLS	Secure Sockets Layer/Transport Layer Security
MQTT	Message Queuing Telemetry Transport
ISO/OSI	Open Systems Interconnection model
OSPF	Open Shortest Path First
IS-IS	Intermediate System to Intermediate System
IP	Internet Protocol
RTT	Round Trip Time
CSV	Comma-separated Values
PNG	Portable Network Graphics

List of Appendices

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

D Experimental Evaluation Additional Data

D.1 Throughput

The Qpid-Dispatch need some time to evaluate the messages and send them to the receiver. In the Figure D.1 we can see the histogram of unsettled messages during the singlepoint throughput test. This charts shows the number off received messages, which are not yet evaluated. Note, that throughput is around 90 000 messages per second.

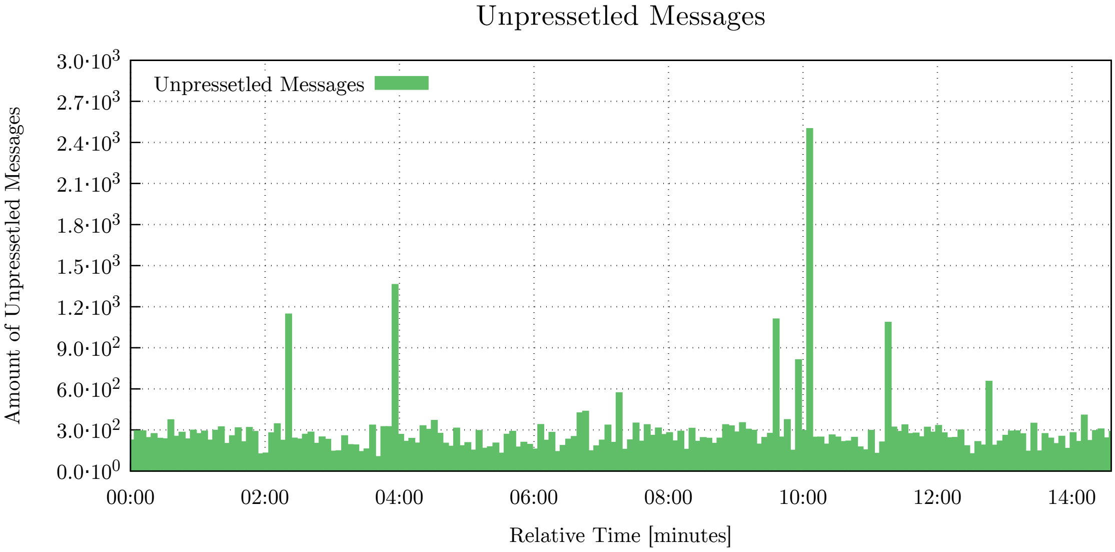


Figure D.1: Chart of the unsettled messages of Qpid-Dispatch during the test.

The flow-control mechanism mentioned in the Subsection 6.1.1 also affected the unsettled message count, which is multiple times higher than in the previous test case depicted in the Figure **??**. The unpresettled message count is depicted in the Figure D.2.

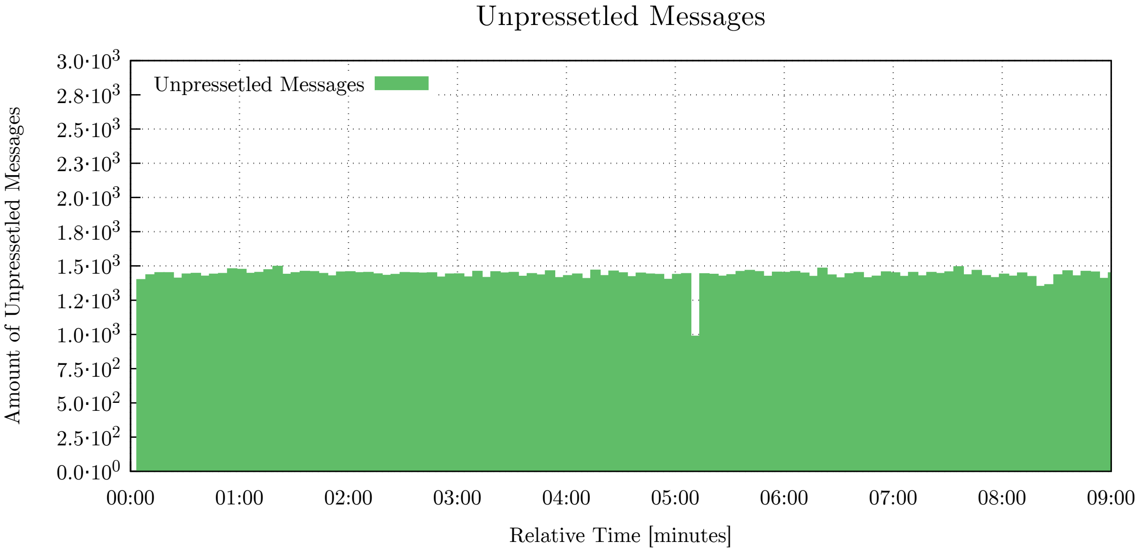
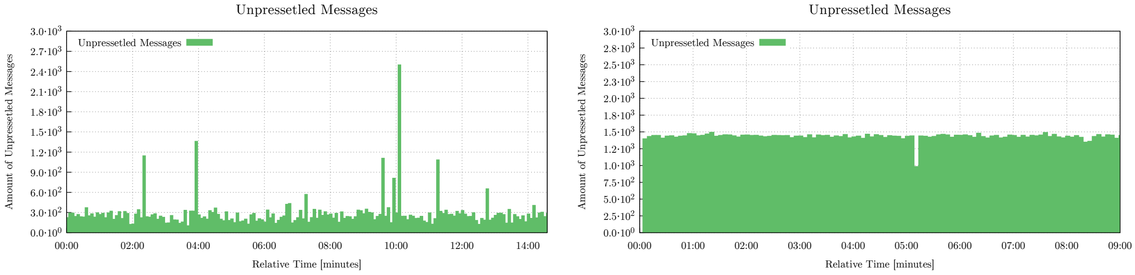


Figure D.2: Chart of the unsettled messages of Qpid-Dispatch during the test.



(a) Topology with a single router node.

(b) Topology with a single Broker node.

Figure D.3: Examples of experimental topologies created for basic performance testing and experiments with Maestro.

D.2 Latency

Unpresettled messages for the router available in the Figure D.4. From the Inspector outputs one can see, that the Broker handled 10 000 000 messages in more than 7 minutes, but the router handled the same amount of messages much faster approximately in 2 minutes and 20 seconds.

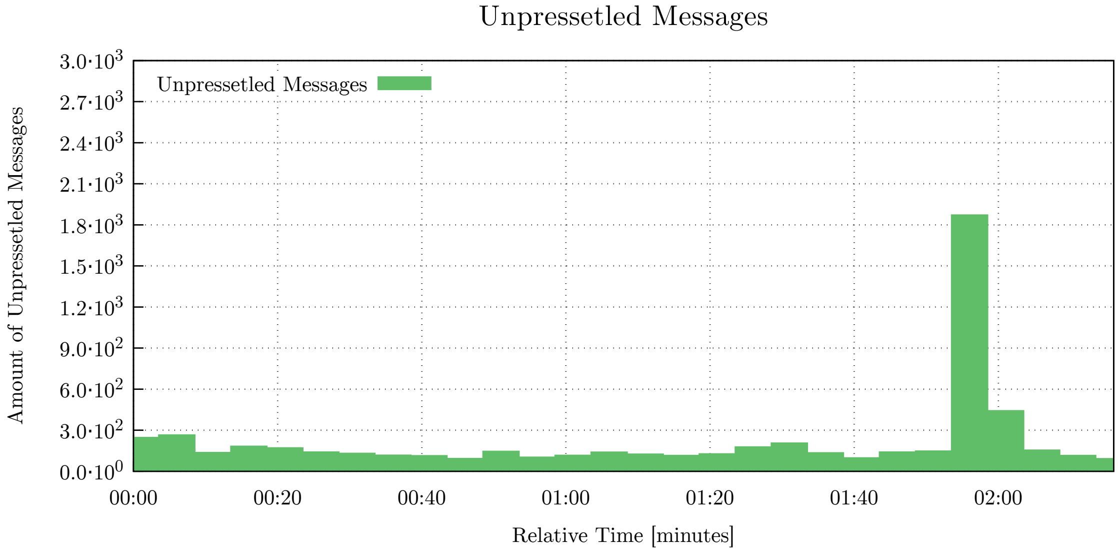


Figure D.4: Chart of the unsettled messages of Qpid-Dispatch during the test.

Since the router applies the flow control during this measurement and the rate is setup to 80 % of maximum, the unpressetled message count is here much lower than in the other cases as it is depicted in the Figure D.5.

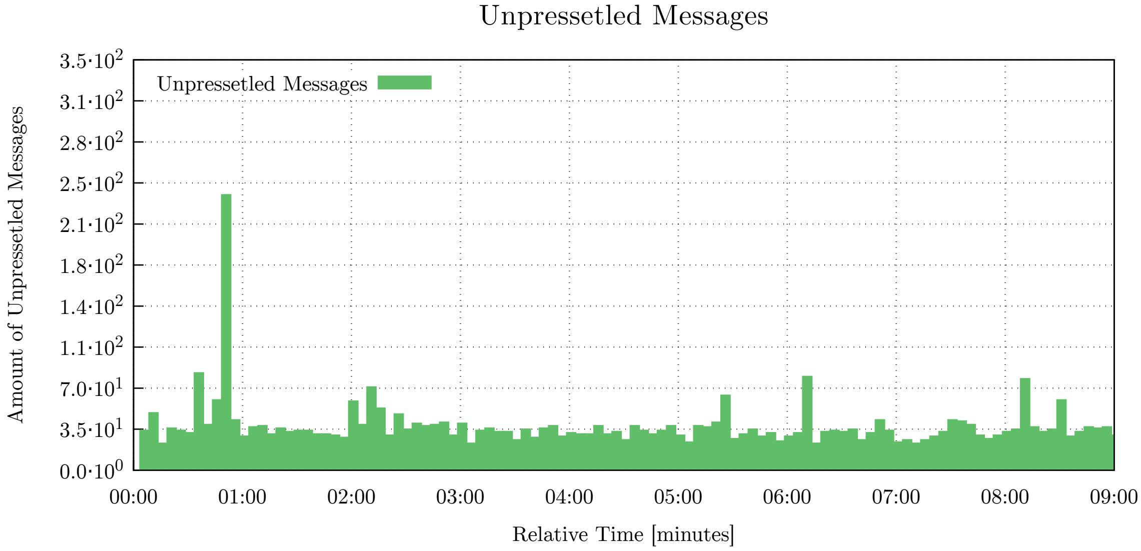
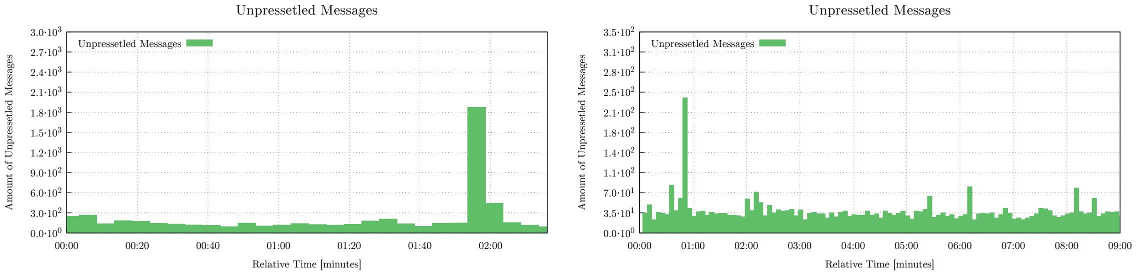


Figure D.5: Chart of the unsettled messages of Qpid-Dispatch during the test.



(a) Topology with a single router node.

(b) Topology with a single Broker node.

Figure D.6: Examples of experimental topologies created for basic performance testing and experiments with Maestro.