Lappeenranta University of Technology
School of Business and Management
Degree Program in Computer Science

Eero Nieminen

MICROSERVICE DATA SHARING USING A LOW-LATENCY INFORMATION BUS

Examiners: Prof. D.Sc. (Tech.) Jari Porras
Ph.D. Kimmo Hätönen

Supervisors: Prof. D.Sc. (Tech.) Jari Porras
Ph.D. Kimmo Hätönen

ABSTRACT

Lappeenranta University of Technology
School of Business and Management
Degree Program in Computer Science

Eero Nieminen

Microservice data sharing using a low-latency information bus

Master's Thesis

2018

73 pages, 30 figures, 1 table

Examiners: Prof. D.Sc. (Tech.) Jari Porras
Ph.D. Kimmo Hätönen

Keywords: master's thesis, microservice, communication, low-latency

This paper studies microservices and microservice communication, with a focus on demonstrating the capabilities of a specific presented concept, that was developed internally at Nokia Bell Labs. This invention aims to work as an information bus between a set of microservices, enabling low-latency communication. The goal of this project, is to implement this invention as an operable prototype, providing a practical proof-of-concept. The aim is to also implement various testing methods on the created prototype in order to measure performance and define limitations in practice.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Business and Management
Tietotekniikan koulutusohjelma

Eero Nieminen

Mikropalveluiden välinen tiedonjakaminen hyödyntäen matalalatenssista tietoliikenneväylää

Diplomityö

2018

73 sivua, 30 kuvaa, 1 taulukko

Työn tarkastajat: Prof. TkT Jari Porras
FT Kimmo Hätönen

Hakusanat: diplomityö, mikropalvelu, kommunikointi, matala latenssi
Keywords: master's thesis, microservice, communication, low-latency

Tässä työssä tutkitaan mikropalveluja sekä mikropalveluiden välistä kommunikointia. Painopisteenä on osoittaa työssä esiteltävän, Nokia Bell Labsilla sisäisesti kehitetyn, konseptin kyvykkyys. Tämän konseptin tarkoitus on toimia tietoliikenneväylänä erinäisten mikropalveluiden välillä, mahdollistaen matalalatenssisen kommunikoinnin. Työn tavoitteena, on toteuttaa tämä keksintö toimivana prototyyppinä, tarjoten käytännöllinen soveltuvuusselvitys. Tarkoituksena on myös implementoida erinäisiä testausmenetelmiä toteutetulle prototyypille, jotta sen suorituskyky ja rajoitukset voidaan määritellä käytännössä.

ACKNOWLEDGEMENTS

I would like to thank Nokia Bell Labs for providing me with this opportunity as well as a hospitable working environment. I would also like to thank Kimmo Hätönen, for providing proficient guidance during this work.

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

BSD	Berkeley Software Distribution
CPU	Central processing unit
DDS	Data Distribution Service
DPDK	Data Plane Development Kit
EAL	Environment Abstraction Layer
FIFO	First-in-first-out
GUI	Graphical user interface
IDE	Integrated development environment
IOT	Internet of things
IP	Internet Protocol
KPI	Key performance indicator
MQTT	Message Queuing Telemetry Transport
N2NQ	Many-to-many queue
NTP	Network Time Protocol
QoS	Quality-of-Service
RAM	Random-access memory
SOA	Service-oriented architecture
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual machine

1 INTRODUCTION

This introduction chapter aims to clarify the basics of the thesis. The chapter has been divided into three areas. At first, the aims and scope of the thesis are presented. This is followed up by an explanation of the used research methods, as well as the core research question, that these methods aim to provide answers for. The chapter concludes with an overview of the structure of the rest of the thesis, in which core contents for each of the following chapters are presented.

1.1 Goals

The goal of this thesis is to study the viability of the distributed many-to-many queue (N2NQ) concept as a method of providing low-latency communication between a set of microservices. The study is executed in the form of an operable prototype that can be run with a set of varying factors to serve differentiating use cases. By operating the prototype based on a comprehensive list of test cases, the practical functionality and performance of the N2NQ concept can be evaluated in a variety of situations.

In order to focus the scale of the thesis into the most useful aspects, a set of delimitations have also been placed. This thesis has been limited to studying the N2NQ concept on a local level as well as a distributed single cloud level. While the possibilities to study the concept on a much larger scale, such as a in a multi-cloud environment, would be intriguing, this has been seen as going outside of the scope of the main goals. This thesis will also focus entirely on the study of the communication methodology, or the N2NQ, and not the microservices themselves. What this means is that, outside of providing testing functionality for the N2NQ, there are no additional tasks assumed for these microservices or what they would require the communicated data for. They are only seen as the start and end points for message transmission.

1.2 Research method

This thesis has been created as a feasibility study, aiming to test the effectiveness and practicality of a pre-proposed concept, N2NQ. To study the concept in practice, a

prototype was constructed. The design of the prototype focused on creating a simple and optimized environment, where the core ideas of the concept could be executed and measured.

The value of the concept was tested from the perspective of quantitative research, based on clear numerical results from conducted tests. Based on the developed prototype, a set of test cases were produced and executed. From these cases, sets of data were gathered and formalized into graphs and models. The developed prototype and the concept of N2NQ itself was analyzed based on these visualizations

1.3 Structure of the thesis

This thesis is divided into seven chapters. After this introductory chapter, the document explains the relevant background information related to the subject of the thesis. This includes general information regarding microservices and how they traditionally communicate with each other, as well as insight into specific technologies and solutions that provide the required capabilities. The next chapter focuses on the N2NQ concept itself, breaking down the original idea and why it was conceived in the first place. The chapter also specifies how this concept is being handled by the thesis itself and exactly what aspects of it are the focus.

Going forward, the fourth chapter explains the practical prototype that was developed for research of the concept. The chapter first goes through the design process and methodology of the prototype and then homes in on the actual implementation, explaining in detail what elements were developed and how they operate. This is followed by fifth chapter, that focuses on testing in all aspects. The chapter explains how these tests were performed, showcases what results they produces and then discusses the significance of the findings. In chapter six, this thesis transitions into discussing the possible directions, where the work produced so far, could be taken into future. Finally, the thesis ends with concluding chapter that summarizes the key points raised so far by the previous chapters.

2 BACKGROUND

This chapter focuses on establishing a wide variety of background information, related to the subject of the thesis. These underlying concepts were seen as important for understanding the fundamentals of microservice communication and the N2NQ concept. The chapter first outlines the basic ideals of microservice architecture and communication. It then focuses directly on how message transmission can be implemented, while also showcasing a variety of already existing technologies from the field. These solutions have attempted to improve message transmission and could be seen as alternatives for carrying out microservice communication. The chapter concludes with a paragraph detailing the communication protocols Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), that are generally viewed as cornerstones of modern message transmission and were integrally related to the realization of the N2NQ prototype.

2.1 Microservice architecture

Microservices are a relatively new style of software architecture, based on some of the ideals of SOA (service-oriented architecture). These ideals include modularity, isolation and distribution, all of which can be seen as core features of microservices. There are, however, also many aspects that distinguish microservices from SOA, and in general, the two architecture patterns should be viewed as separate. [1, 2]

While no common consensus on the definition of a microservice exists, it could be seen as a small, independent process, providing a specific service, that interacts with other processes using messages. Unlike the services in SOA, microservices can be deployed and operated separately from other services, achieving true independence. To achieve these autonomous operations, each microservice should include an operating system (OS), runtime components and related dependencies. The aim of microservice architecture, is to develop distributed software applications, using only these independent microservices as modules. This collection of microservice modules is designed and coordinated to communicate with each other in order to perform more complex functions and provide a wider set of services. [1, 3]

In many ways microservice architecture could be seen as an alternative to the monolithic architecture, a traditional style of software design in which applications are built as singular entities. While monolithic software can be simpler to implement and encapsulate, they also tend to face a multitude of issues, especially with large-sized applications. These issues stem from large complexities, limited scalability and strong dependencies, that are usually found within monolithic applications. Problems like these can make complex monoliths challenging to deploy, maintain and, especially, to build upon on the long run. [4, 5]

There are many benefits that can be achieved when choosing to utilize microservice architecture. Since the microservice modules are designed to be independent, they can also be developed separately using different languages and paradigms. As long as the modules can communicate using messages, their internal logic can be hidden from the rest of the application. This also leads to another obvious advantage in the form of added changeability, scalability and fault tolerance. Improvements can be made exactly where they are needed, without disturbing the larger applications as a whole, if properly planned. [4, 5]

2.2 Microservice communication

It is traditionally agreed that each microservice should be designed to be as independent as possible. This is contradicted by the fact that when designing a larger application based on the microservice architecture, individual services require data from other services so that larger tasks can be performed. The paradoxical nature of microservices architecture makes it difficult to define a commonly agreed upon solution for best practices. As such, there are no industry standards in the field to be found, similarly as with the lack of a common definition on the microservice term itself. Regardless, the distributed nature of microservices prevents an application from simply managing operations with functions calls within a process. Network communications are required, but how those should be executed is not entirely obvious. [6, 7]

While the used methodology differs between implementations, asynchronous messaging seems generally preferable to synchronous. Traditionally in synchronous communication,

the process sending the message requires the receiver to send back a response to confirm that the message was successfully sent. The problem with this is that in most cases the sending process becomes directly dependent on the receiver and cannot proceed before the confirmation has been received. This forms dependencies between the processes, which conflicts with the core ideal of microprocesses requiring independence. Asynchronous communications, on the other hand, are able to avoid this problem altogether. [7]

With asynchronous communications, message sending processes do not wait for any responses back from the receiver in order to continue operations. In many implementations, no responses are sent back at all. This means, that the processes can operate on their own, without dependencies to each other. This independence can be further improved by implementing a mediating solution, like a message broker, to handle the data delivery. This way the receiver can send the messages to the mediating middleware and let that handle the necessary routing. This helps the messaging processes to remain completely oblivious of each other. [7]

2.3 Message transmission

Defining the ways in which processes connect and communicate with each other is a crucial part of any system and network architecture. This thesis mainly focuses on three traditional messaging patterns for data distribution, all of which have been viewed as usable in messaging with microservices. These are: request-reply, message queueing and publish-subscribe. [3]

Request-reply is a two-way messaging pattern, in which two participating processes communicate to each other, with one being the requester and the other the replier. The communication is handled with one of the processes sending a request message to the other and then waiting for a response. After the other process has received the request, it replies with a message of its own, usually containing the requested data. [8]

Utilizing a request-reply traditionally produces a system with synchronous messaging, where the requestor blocks other functionality while waiting for a response. If multithreading is an option, another possibility is to have a separate thread listen for

replies, while allowing other threads to retain asynchronous functionality and make requests. This also allows for an application to forward multiple concurrent requests. It is, however, debatable whether the use of multithreading coincides with the ideals of microservices, which aim to lightweight and independent. [8]

Message queueing is traditionally used for point-to-point messaging between two processes, or between two threads in a single process. In it, a queue is created with interfaces for both interacting processes, with one of the processes storing messages into this queue and the other one retrieves them. The two communicating processes do not need to interact with the queue simultaneously and are free to send and retrieve messages as they wish, thus maintaining asynchrony. This assures that asynchrony is achieved between the processes. When a message has been retrieved, it is removed from the queue and the next message becomes available. [9]

A single message queue can also be utilized between multiple processes for input and retrieval, but the same rule for message removal applies. This means that each message is still being received by only one process. Traditionally, the nature of the queue necessitates that only one process is able to write and to read from the queue at the same time. This can cause some issues with congestion when utilizing multiple processes, although operations can still be established utilizing locks that prevent simultaneous access. While these kinds of operations can lead to performance drawbacks, they can also improve system reliability. Due to the decoupled nature, even if one process fails, others could optimally still continue operating. Regardless, the publish-subscribe pattern is often viewed as a more suitable option for multi-process communications. [9]

Publish-subscribe pattern is a messaging solution that allows for a set of processes to communicate with each other while retaining loose coupling. In it, the producers of data are referred to as publishers and the receivers of data are subscribers. Traditionally in a publish-subscribe system, messages are being transmitted and filtered through an intermediary message broker or bus. The publishers transmit messages to the broker which are then classified either based on a predetermined set of topics or by their contents. The receivers subscribe to the broker based on these topics or by specific content

characteristics. Messages are then filtered and forwarded for each correct subscriber and only them. [10, 11]

Unlike traditional message queues, publish-subscribe solutions are naturally designed to serve a multitude of processes, both publishers and subscribers. Whether these communications are synchronic or not, depends on the solution. While the different processes, by nature, remain oblivious of each other, their connection to the broker can be managed in different ways. It is possible, for example, to have the producers publish asynchronously, while managing the consumptions of messages in a synchronous manner. [11]

There are a few ways how the message removal can be implemented in a publish-subscribe system. Instead of removing messages after each receival, they are traditionally deleted after a set amount of time has passed. During this time, a message should be freely accessible for each subscriber. Another possibility is removing messages only after they have been consumed by all of the subscribers, although this assumes that each subscriber would be interested in every message, which is not always the case.

Each of these presented messaging patterns, or at least some of their ideals, were seen as corresponding with the different use cases of the N2NQ concept, to some extent. As such they were seen as very influential on the design of the developed prototype. In addition to these, a set of ready-made messaging solutions were also studied for the thesis. The most significant of these, Apache Kafka, Message Queuing Telemetry Transport (MQTT), Data Distribution Service (DDS) and Data Plane Development Kit (DPDK), are presented next since they were used as inspiration and influence in the study and creation of the thesis.

2.3.1 Apache Kafka

Apache Kafka is a distributed platform, designed to offer stream processing capabilities, that operates as a publish-subscribe message broker. The system was originally designed and developed by engineers at LinkedIn, in order to deliver large quantities of event data to a diverse group of subscribers. Due to its effectiveness, the technology was implemented to operate as the central pipeline system for LinkedIn's online data. Since then, Kafka has

become a well-known solution around the industry and is currently being managed by the Apache Software Foundation. Apache Kafka was developed using Scala and Java and is currently being utilized by a large amount of notable enterprises, such as eBay, Netflix and PayPal. [12, 13]

As a publish-subscribe messaging solution, Kafka works to decouple publishers of data from the subscribers. Kafka operates as a message broker between the data sending publishers and receiving subscribers, allowing for the subscribers to request data related to them, without requiring any knowledge of who published it in the first place. This decoupling also applies to the publishers, which only focus on delivering data to the broker. The broker then makes sure that the correct information is being transported to the consumers that subscribed for it. In the case of Kafka, the correct routing of data is handled with a topic-based system. [13]

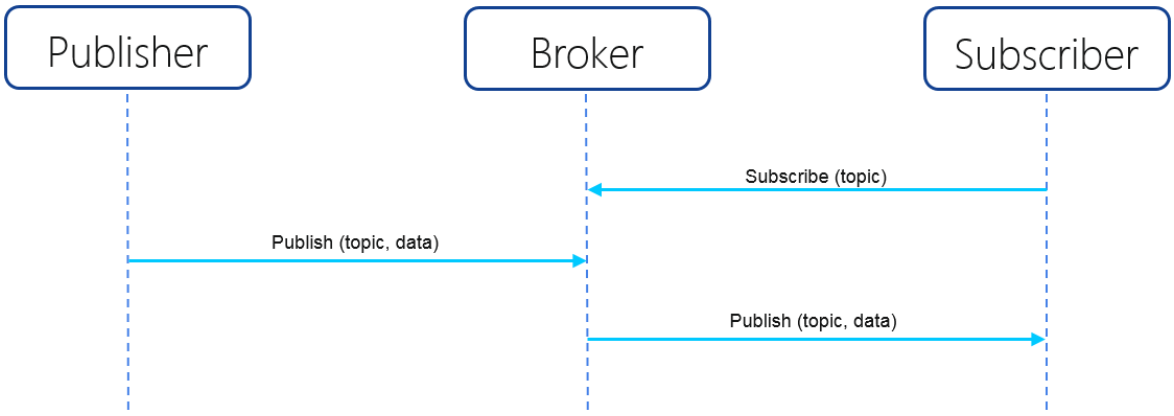

Figure 1. Topic-based publish-subscribe communication model [10].

In a topic-based system, published data is organized based on keyword identified topics, to which each of the consumers are able to subscribe to. Data messages can then be distributed to the correct consumers based on these subscriptions. [10] A visualization of the topic-based communication model can be viewed in Figure 1 above. Data within Kafka is managed with a time-based system, where messages are available for a predetermined amount of time, measured from their introduction. After this time has passed, messages are removed from memory. [13]

As a competitive publish-subscribe broker, Kafka provides specific benefits for its correct usage. These benefits include high scalability as well as high availability, with the ability to detect faults and re-route functions to other services, so that operations are not interrupted. There are no natural bottlenecks in the overall Kafka architecture or any of its layers. In terms of supporting a large microservice architecture though, Kafka on its own can be viewed as lacking. It is possible that maintaining connections and large amounts of data can become a challenge, although these issues can be mitigated by introducing complementary solutions, such as different APIs. [5]

Message brokers, and Apache Kafka especially, were seen as a significant point of influence when studying the topic of this thesis and microservice communication in general. While providing suitable inspiration and possibilities for learning, no existent message broker was in practice utilized in the creation of the actual thesis work itself.

2.3.2 MQTT

MQTT is a protocol designed to provide publish-subscribe-based messaging functionality for machine-to-machine connections, while achieving bandwidth-efficiency and low power requirements for the utilized system. Originally developed in 1999 by Doctor Andy Stanford-Clark and Arlen Nipper, employees of IBM and Eurotech, which was then known as Arcom. The main aim of MQTT was to be as lightweight as possible and operate in environments with limited network bandwidth, high-latencies and multiple remote locations. The protocol has been designed to decrease bandwidth use and increase reliability while operating on devices with limited available resources. [14, 15]

As with many other publish-subscribe models, MQTT utilize a broker to direct and distribute data from publishers to subscribers, based on a set of topics, much like the previously presented Apache Kafka. Due to its design characteristics, MQTT has been found to being well suited for Internet of things (IoT) systems, where a variety of small connected devices need to communicate with each other. The light weight of MQTT allows for publish-subscribe messaging to be employed on a multitude of platforms, that other solutions might find too constrictive in terms of available resources. [10]

There are, however, deficiencies that have been observed in MQTT as well. MQTT was originally designed to be implemented in fixed architectures, with a set quantity of producers and consumers of data. This can lead to MQTT having troubles maintaining mobility and coping with disruptions, leading to instances of data loss when recovery is not possible. In a situation with dozens of devices, and especially with the recent emergence in popularity and usability of microservices, MQTT can prove to be too inflexible. While there have been proposed solutions to overcome these kind of issues, many of them are new and have been tested relatively little in practice. [15]

As with Apache Kafka, MQTT was never directly used in the creation of the thesis prototype. In terms of this thesis, its purpose remained purely educational on different alternatives that others have developed in the past. This being said, many of the characteristics of MQTT, especially lightweight resource requirements, were also deemed advantageous for the executed thesis work.

2.3.3 DDS

DDS is an Open Management Group (OMG) approved machine-to-machine middleware standard, which offers scalable, high-performance data-centric communication solutions using the publish-subscribe pattern. The specification was originally developed starting from 2001 in a collaboration by the American Real-Time Innovations and the French Thales Group, with version 1.0 being published in December 2004 and the newest version, 1.4, in April 2015. Since its inception, the standard has been deployed on wide scale especially in the fields of defense and industrial applications, with embedded, real-time systems in mind. [16, 17]

As with other publish-subscribe solutions, DDS operates as a middleware between specific network processes, receiving and sending data messages from point to point. DDS implements a topic based publish-subscribe pattern, allowing subscribed processes to receive exactly the data they require. As a data centric solution, DDS is first and foremost concerned with the value of data and making sure that each of the connected processes share this understanding as well. Another important characteristic of DDS is carrying out interoperability between services created with different languages and platforms. [16, 18]

Unlike many other publish-subscribe solutions, DDS does not utilize any traditional message brokers. Instead, all publishers and subscribers connect to a virtual "global data space", that simulates a shared local storage of data between the processes. In actuality, this simulated shared data space is just a collection of local data storages from different connected processes, all of which store locally only the data they require. From the perspective of the connected processes, the global data space appears as a part of the native memory to which they can write and read data. The data space itself operates as a sort of data bus, managing the connections and flow of data from publishers to subscribers, based on defined topics as well as an expansive set of Quality-of-Service (QoS) parameters. These QoS parameters can be seen as a collection of characteristics, that guide the behavior of a specific service to a desired direction. [16, 17, 18]

DDS provided an interesting comparison point with Apache Kafka and MQTT, due to it employing the publisher-subscribe pattern, without the utilization of a traditional message broker. This provided valuable insight during the planning and background research process, regarding alternative methods of achieving successful communication solutions between processes. Regardless, DDS was not directly used in creation of the thesis prototype, same as with its aforementioned contemporaries.

2.3.4 DPDK

DPDK is a collection of libraries for data plane and drivers for network interface controllers, designed to improve the speed of packet processing. Originally developed by Intel in 2010, DPDK is an open source project with a Berkeley Software Distribution (BSD) license, which is currently being managed by the Linux Foundation [19]. The technology was originally designed to operate with Intel x86 but is now compatible with any processors. [20, 21]

DPDK is mainly used for development of data packet networking applications, with a focus in achieving high speeds on Linux OS. The key to speed in DPDK is the way it allows for packets to bypass the kernel space, or the core of the OS, in its entirety. As presented in Figure 2, data packets must traditionally go through the Kernel before

reaching destined applications, but with DPDK this, and additional time requirements related to it, can be avoided. [20, 21]

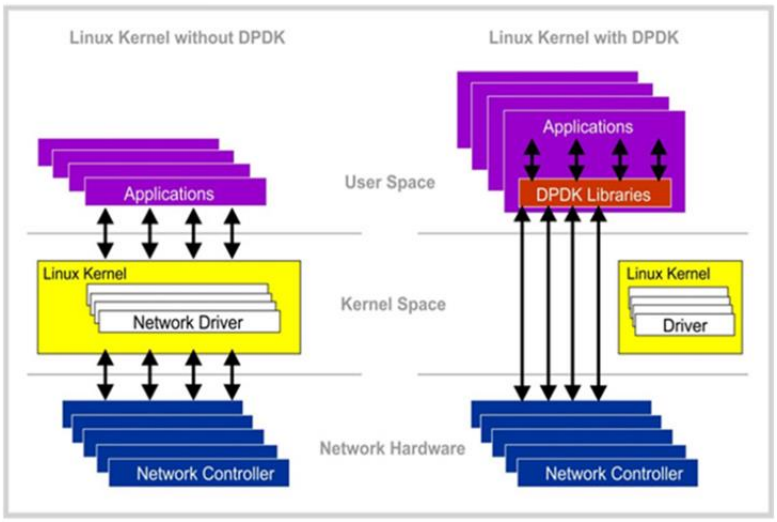

Figure 2. Linux Kernel with and without DPDK [20].

DPDK achieves this kernel bypass with the use of the Environment Abstraction Layer (EAL). EAL is formed from a set of libraries that allow the local hardware and software specifics to be hidden for the applications. These applications are then offered an interface for specific library functions, all of which are executed in user space. DPDK also utilizes a set of specially designed, high performance focused, components. These include specialized memory and buffer managers for utilization of a huge page memory and reducing buffer allocation time. DPDK also utilizes a safe lockless queue manager, and poll mode drivers, which means that no scheduling is supported. This means, it possible to avoid interrupts that waste the attention of the central processing unit (CPU) and lead to overhead. [20, 21]

The possibilities of utilizing DPDK with N2NQ were considered during the early stages of the research process but were eventually left out of the thesis. While possible benefits were recognized, their implementation would have pushed the scope of the thesis out of the essential core elements for N2NQ.

2.4 TCP and UDP

TCP and UDP are a pair of network communication protocols, used for transmitting packets of data over the Internet and local networks. Both of them can be seen as extensions of the Internet Protocol (IP) protocol, targeting specific IP addresses as destinations, and are core members of the Internet protocol suite, commonly known as TCP/IP. While both TCP and UDP are closely related, they also possess distinct differences in abilities and use cases. [22]

TCP is stream oriented communication protocol. This means, that packets get sent in steady data streams of bytes, without distinguishing between messages. Before any packets can be sent, TCP requires a connection to be established between the two communicating processes. This connection must be upheld throughout the transfer process. Due to utilizing a continuous connection, data packets can be sent to either directions. This means that only one connection is required to establish a dialog between two points. TCP has been recognized as a widely accepted and utilized communication protocol, being relied upon by significant networking applications, such as email systems as well as the aforementioned MQTT. [22, 23]

The main benefit of TCP is its ability to produce accurate and reliable results. TCP has the inherent ability to provide error checking and can retransmit packets if they turn up faulty in the destination. TCP protocol is also able to control the flow of data in order to relieve congestion related issues. As a result of these security features, TCP is viewed as a more heavy-weight protocol than its counterpart, that sacrifices some of its speed for additional reliability. [22, 23]

UDP, on the other hand, is a significantly simpler protocol. Instead of utilizing steady streams, UDP transfers data as individual packets, which are referred to as datagrams. UDP also does not require any continuous connections to be established for data transfer. This means, that the transfer is always one-way only and the source never waits for any confirmation to come back from the destination. Because of this, UDP does not offer any kind of inherent guarantees for packet deliver or duplicate protection. As such, achieving constant reliability is difficult. Packets can get lost or corrupted in the way to the

destination, without the source ever finding out. [22, 23]

Despite the risks related to reliability, UDP also benefits from its simplicity. Since UDP never waits for the destination to respond, there is much less overhead involved. This allows for faster communication rates and UDP is, in general, viewed as a faster alternative to TCP. Due to its characteristics, UDP is traditionally used in situations where minor data loss is viewed acceptable and speed cannot be compromised, for example, online gaming. UDP is also more suitable when the application requires data to be broadcasted to multiple locations simultaneously. [22, 23]

In the case of this thesis, TCP was ultimately chosen as the communication protocol for the development of the N2NQ prototype. This was decided for multiple reasons. Despite UDP's advantages in speed, the reliability offered by TCP was seen as a more important benefit for this particular situation. Even though achieving low-latency was chosen as the main key performance indicator (KPI) for the thesis, it was equally important to be able to reliably measure all of the messages sent. TCP's ability to offer two-way communication was also noted as a marked advantage over UDP, as some of the functionality, implemented in the prototype, relied upon it.

3 N2NQ

This chapter details the core concept of N2NQ, which formed the starting point for the entire thesis. At first, the initial problems concerning microservice communication are presented, as these are seen as the original motivation for the N2NQ concept. Following this, the ideology and design of N2NQ itself is described in detail. This also includes the main use cases, that N2NQ is aimed to work in. Finally, the chapter concludes with a summary of the specific N2NQ details that this thesis utilizes and studies.

3.1 The problem defined

With every advance in networking technology and standards, the amount and complexity of network elements have continuously increased. With the approach of 5G networks, the way networks operate is being redefined once again. Due to the explosive rise of social networking and IoT, more devices than ever are communicating with each other, leading to massive growth in mobile network traffic. The sheer volume of data and services forces developers to focus more on distributed computing. From this point of view, the increasing utilization of microservices makes sense. However, the effective use of microservices also requires the ability to share data fast throughout the entire network. Especially in 5G networks, achieving and maintaining low latency communication can become a necessary but challenging task to achieve. [24, 25]

Many of the current data sharing solutions are not optimized for low latency in large networks. Delays nearing one second or more might be acceptable in basic messaging solutions but are entirely unacceptable in 5G networks and systems based on microservice architecture. Another issue with traditional point-to-point communication is the requirement for a large number of standardized interfaces. When creating a microservice system, the amount of various kinds of independently working services drastically increases. This forces developers to look outside of developing unique interfaces for services, as the result would easily become too complex and difficult to manage. [26]

Other issues can also arise from the basic requirement of network services having to connect to each other. When each service needs to communicate with all the others, the simplest solution is to send each message everywhere. This will undoubtedly lead to massive amounts of data replication. A significant part of this replication can be seen as unnecessary, since in most cases not every one of the messages are relevant for every service in the network. Due to the ever-expanding amounts of data, eliminating any unnecessary traffic would be beneficial. While there have already been ways to mitigate unnecessary replication of data with current communication methods, such as filtering used by many publish-subscribe brokers, these have usually also lead to additional amounts of processing power required and overhead [27].

To truly get the benefit out of microservice architecture in 5G networks, these aforementioned issues need to be solved. A solution is required, that is capable of achieving low latency communications with large quantities of services, manages to keep standardized interfaces to a minimum and avoids unnecessary data replication, without excessive processing power requirements. This is where N2NQ is aimed to provide benefits.

3.2 The original concept

N2NQ is a fast information bus concept, developed internally at Nokia Bell Labs, and generated into a formal description of an invention. The main goal of N2NQ is to offer low-latency data sharing capabilities to a large number of microservices, both on a local and a wide spread level. The core functionalities of N2NQ are to gather packets of data from a set of sending network functions, store them into an internal queue and then share them forward to the correct network functions, that are interested in receiving them. The main goal is to achieve all this as fast as possible.

What differentiates N2NQ from other communication systems is its lightweight and straightforward functionality. Once packets have been received by the N2NQ and stored into a queue, the system immediately starts sending them forward to available receivers. To avoid unnecessary operations, the N2NQ only stores data packets that are in a transition state between implemented and available network functions.

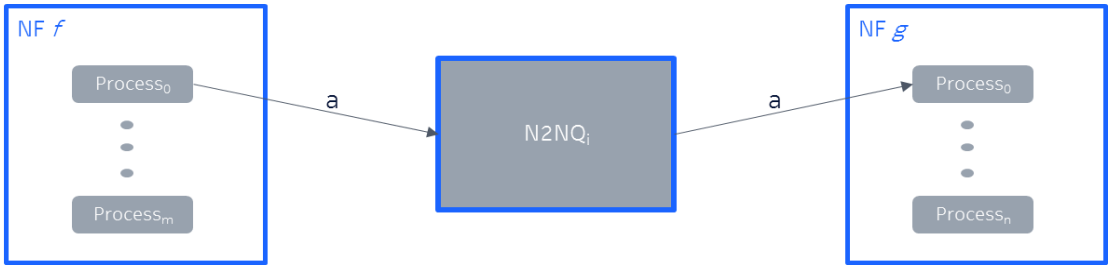

Figure 3. A basic set-up of N2NQ.

In its most basic form, N2NQ acts as a local data bus that stores and transmits messages between network functions within a single cloud environment or even within a single machine. This simple set-up can be seen in Figure 3. In a setup like this, there would only be two interfaces that need to be implemented: one for insertion of data and another for retrieval. As far as senders and receivers are concerned, both services need only to interact with one of these interfaces. In the most basic implementation, each packet would get sent forward from the queue in a first-in-first-out (FIFO) order. However, this order could theoretically be improved with additional instructions or requests made by the receiving network functions.

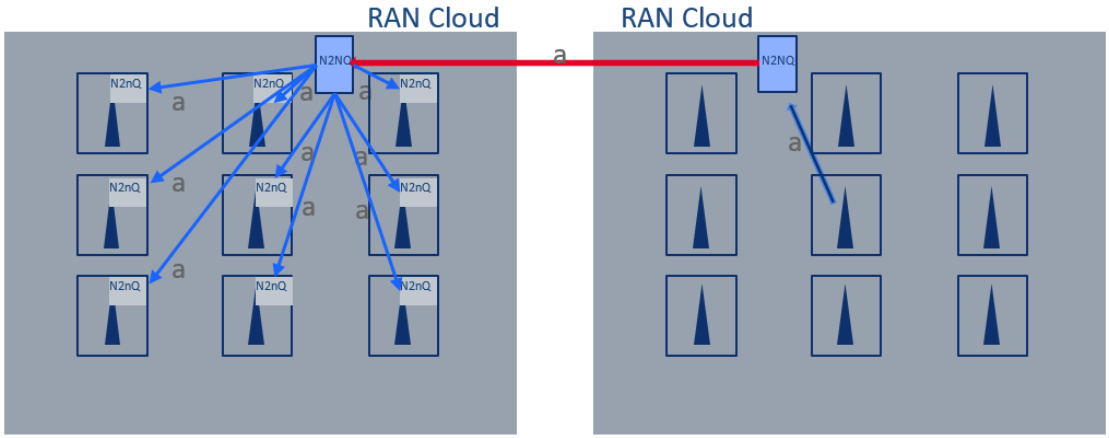

Figure 4. A complex set-up of N2NQ.

In theory, N2NQ could also be implemented as a more complex set-up that communicates between different clouds. In this case, each of the communicating clouds would have a central N2NQ instance which would interact with all of the network functions found in that specific cloud. These central instances could then also share data with other similar N2NQs found on other clouds. This way the scope and usability of N2NQ could be increased to a

much larger scale. Even in this expanded multi-cloud version, N2NQ would only require a set of four standardized interfaces in total. Two for receiving and sending messages to local network functions and two for doing the same with other cloud N2NQ instances. This way complexity of the communication system can be kept on a relatively low level, even if the entire network encompasses a wide area with a substantial amount of microservices. A visual representation of this set-up can be seen in Figure 4.

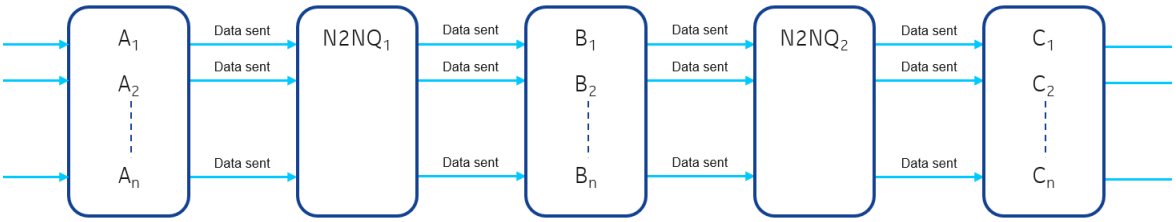

Figure 5. N2NQ used as a load balancer.

There are also other specific use cases where the N2NQ could be utilized. For example, N2NQ could be specifically implemented to work as a load balancer, operating as a group of instances between a set of network functions. A draft of this use case can be seen in Figure 5. In this example, each of the presented network function groups, A, B and C, are formed from a set of identical microservices. The task of these groups would be to process data and send it forward in packets as fast as possible. This data would then be received by the adjacent N2NQ, which would in turn send each packet forward to the next free service in the following group of microservices. This series of microservice groups and N2NQs could be repeated as many times as necessary, until the data would be refined enough.

There would be no matter which of the specific microservices would receive which packet of data, since they would all execute the same processes, working towards a common goal. In this situation, the N2NQ could operate using the ideals of the message queueing pattern, with each message being sent only once and then removed from the queue. Even with multiple points of entry and retrieval for the queue, asynchrony could be maintained between the microservices, if managed correctly.

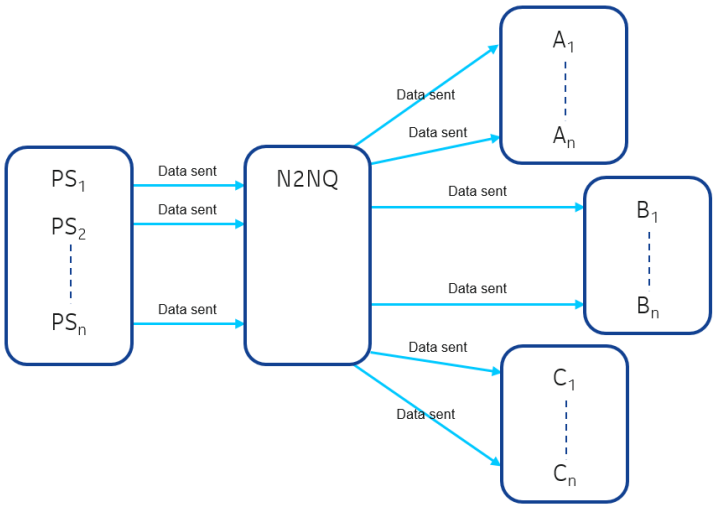

Figure 6. N2NQ used as a broadcaster.

N2NQ could also be implemented to act specifically as a broadcaster, if it was deemed necessary that every network function would require every packet of data. A sketch of this use case has been outlined in Figure 6 above. In this situation, the main goal for the N2NQ would be that each of the receiving services would receive every packet of data, without exceptions. This use case would obviously result in a lot of data replication and additional processing requirements, negating some of the natural benefits of the concept, but could still be applied if necessary. In a situation like this, management of the queue itself would operate closer to the publish-subscribe pattern, in certain ways. Making sure that the messages do not get removed prematurely could be prevented by removing them based on a set time. It would also be viable to remove messages when all of the receiving microservices have collected them, since all messages would be distributed evenly to everywhere.

Naturally, these are not the only ways in which N2NQ could be deployed or operated. The functionality could also be enhanced by allowing the receivers to request for specific data points from the N2NQ instance. Based on a maintained index of data attributes, the instance could be able to respond directly to the requests made by receivers, effectively implementing request-reply functionality to the system. Other factors, such as data point ordering and removal, could also be finetuned based on the requirements of the specific situation.

3.3 Significance for this thesis

No prior practical studies on the capabilities of N2NQ have been established prior to this thesis. The goal here was to create an original proof-of-concept in the form of an operable prototype. As an ambitious concept, not every element of N2NQ can be implemented and tested in a single thesis, and since this is the first practical approach on N2NQ, the aim here was to limit the concept to its core elements. This meant implementing a simple practical solution and testing how it performs under a rudimentary set of options. During the thesis work, many additional possibilities for research were discovered and discussed, although, a majority of these were never implemented or tested during the thesis work. Many of these, however, are presented later in the thesis, as possible future directions for continuing research into the topic.

Concerning this prototype, N2NQ was implemented as a local information bus, operating in two distinct setups: a singular virtual machine (VM) and a collection of three VMs, located within a single cloud environment. In addition to the N2NQ itself, a set of additional processes, sending and receiving data packets, were also developed to work as a stand in for proper microservices. On a development level, N2NQ was designed solely based on the visions raised in the original concept paper, without the utilization of any already existent communication solutions.

4 OVERVIEW OF THE PROTOTYPE

To demonstrate and test to core functionality of the N2NQ concept, a prototype had to be made. This part of the report details the design and creation process of the prototype in detail. The chapter first focuses on the design of the prototype, covering the general ideology, requirements, measurements and architecture. Following this, the focus turns to the actual implementation of the prototype, first covering the main elements and then the ways in which these communicate with each other. The chapter ends by presenting the graphical user interface (GUI), that was developed for operating and testing convenience.

4.1 Design of the prototype

The goal of the prototype was to create a basic level proof-of-concept of the N2NQ concept. To achieve this, the following set of elements had to be implemented:

1. A set of sending microservices that connect to the N2NQ, generate messages based on an algorithm and send them to the N2NQ.
2. A set of receiving microservices that connect to the N2NQ and accept messages transmitted from it.
3. The N2NQ itself, which receives messages from the sending processes, stores them to a queue and sends them forward to the correct receiving processes, based on predetermined criteria.

While additional functionality was also developed, these three elements became the core foundation for the prototype and the rest of the design focused around achieving their execution and usability. A more comprehensive description of each element can be read further down in the report.

The prototype was also designed around three distinct use cases, all of which required the implementation of varying functionalities, in addition to the core elements. Each of these use cases were designed to implement and test different conditions for the N2NQ concept. These use cases were:

Use case 1: Each data point should be removed from the queue only after a set amount of time has occurred from their inclusion. This means that each message would be shared to every receiver, effectively turning the N2NQ into a broadcaster.

Use case 2: Each node of data should be removed from the queue after they have been sent to a receiver. In this case each message would be sent to only one receiver, limiting traffic and utilizing the N2NQ a sort of load balancer.

Use case 3: The receiving processes should be able to request more precisely what kind of messages they require. This way specific conditions could be introduced for each run and the receivers would gain more control for the runtime.

In addition to these core elements and use case functionalities, a set of testing functionalities also needed to be implemented, so that the performance could be measured. A more detailed description of these can be found later in the next chapter of the document.

The scale of this prototype was initially set on a fully local level, with all of the prototype processes running on the same Ubuntu operated VM. This way the focus could be maintained strictly on the development of the core functionality on a stable platform. Further in development, however, this was expanded into a distributed set up with three VMs running in a single cloud environment.

4.1.1 Architecture of the prototype

As previously defined, the core of the prototype was designed over three distinct elements, the sender microservices, the receiver microservices and the central N2NQ. The general idea was to design and implement each of these parts as individual and unique programs that could all run in separate processes, while also being able to communicate with each other. The senders would generate messages and push them to the N2NQ for temporary storage. The N2NQ instance would then process the messages and send them to the correct receivers. This basic model of architecture has been visualized in Figure 7 below.

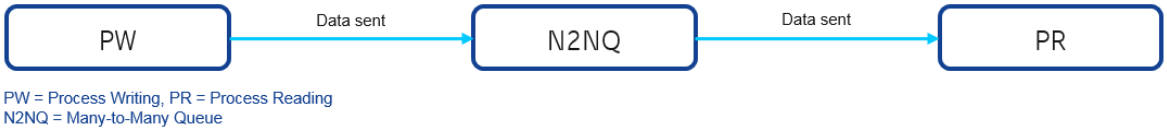

Figure 7. The basic architecture N2NQ.

In this basic model, the senders and receivers were designed to function only as the start and end points for each message, with their parts limited to the basic messaging functionalities. While the senders and receivers would represent the part of microservices in the N2NQ concept, it was never important what they would require this information for, in the scope of this thesis. Furthermore, the contents of the messages were only valued for its functionality and did not contain any further significance or value. For this reason, no additional functionality for the senders and receivers was designed, unless it directly served the communication system or the testing capabilities of N2NQ. On the other hand, the N2NQ instance itself was always designed to be the central focus of the prototype, with the sender and receiver microservices providing methodology for executing and testing its capabilities.

4.1.2 Key measurements

The main measurement for the prototype was defined as latency. This meant measuring and collecting the time it took for each message go through the system, from the senders to the receivers. This was chosen as the single most important metric, as achieving low-latencies was deemed as a key goal for the overall N2NQ concept. Another measured metrics collected was the delay between each sent message. While it was decided early on, that each message would be sent with a small controlled delay, there was also an additional small time loss to be expected. By measuring this small delay, the stress of the system could be evaluated.

In addition to these calculations, it was deemed necessary to provide additional input for the user, in the form of message counters, both in the sending and receiving end, as well as average transmission rates. By measuring and showing these in real time on the GUI, the current state and progress of the prototype could be observed during runtime.

4.1.3 Tools and techniques

The prototype was built on an Ubuntu Linux 16.04 VM, running on a Windows 7 laptop computer. While earlier versions were also designed to run on this platform, the final prototype was transferred into three identical VMs, running in a shared cloud environment and operating with CentOS Linux 7.4. Each of these VMs utilize an Intel Xeon E5-2630 CPU with 4 active cores, running at 2.3 GHz each and 4 GB of system random-access memory (RAM). Each of these VMs were designed to serve one of the three core elements of the prototype, the sender microservices, the receiver microservices and the N2NQ itself. In addition to this, the VM holding the N2NQ instance could also be used for running all of the elements concurrently.

C++ was chosen as the most suitable programming language for the prototype, since it provided object-oriented capabilities and was a familiar language to work with. The core functionality of the prototype was built using NetBeans IDE 8.2 (integrated development environment). This provided a familiar and flexible environment for C++ programming. The prototype was first built with only the core functionality in mind, focusing on a command-line based solution. After the main elements were deemed ready, a GUI was built on top, for increased convenience on runtime. During this, the development also moved from NetBeans to Qt Creator 5.9.1, due to its natural support for graphical development requirements.

4.2 Implementation of the prototype

As presented previously, the prototype was divided into three distinct elements, the senders, the receivers and the N2NQ itself. These formed the core of the prototype and everything in the prototype was designed to serve their functionality. The following subsections detail the developed functionalities of these three elements and explain how they were each designed for use in the prototype.

4.2.1 Senders

A sender process acts as a beginning point for the message distribution functionality in the N2NQ prototype. The senders generate a predetermined number of messages and forward

them to the N2NQ in the form of a TCP data stream. The core of the message is made up from a set of letters, the quantity of which are either generated randomly or set beforehand by the user. In addition to the core data, the messages are also packaged with a variety of header information, including the message size, time of sending, name of the sender and an index identifier, which is unique for every message. All the information regarding each message is also saved in a separate text file as they are sent, by each of the implemented senders. As with the length of the messages, the amount of them being sent is also specified by the user for each run.

Senders have been implemented as a group process, where the different senders have been organized as a flexible sized group of threads. This was done mainly for the purpose of simplifying the structure of the prototype, as all senders could be started through a single process. The number of senders being used for each run is not fixed and can be changed freely by the user, before a run is started. Each sender is designed to connect to the N2NQ with a different port number, to avoid overlap. Once a connection has been established and confirmed, each sender transmits messages with a delay, that can be chosen by the user, without waiting for any responses or requests from the N2NQ. When each of the senders have created and forwarded the desired number of messages, they all generate and send a special ending message, which lets the N2NQ know that the transfer has ended, and the connection can be terminated.

4.2.2 Receivers

The receivers function as the end of the message distribution process. Their main purpose is to receive data messages from the N2NQ and parse their contents. The receivers also count the latency for messages, based on the send time, gathered from each received message's header information, and the receive time, which is checked internally for every message as they arrive. As with the senders, each individual receiver also records the contents and details of each message, with the addition of measured latency, to a specific text file for further use. Also, like the senders, the receivers work as a single unit process, with multithreading implementation for each individual receiver. The number of receivers is fully customizable by the user, although they all communicate with the same N2NQ instance. While receivers have no control over the way senders operate or how many

messages they push forward, they can modify the output they receive from the N2NQ instance.

The receivers differ from the other elements in the way that they have the ability to send messages back into the other direction, in this case to the N2NQ, transforming the connection into a two-way communication. These return messages are used in two distinct ways, requests and responses. Requests are used when the user wants the N2NQ to send messages only when specifically asked by a receiver. These request messages can also be tailored to transmit specific commands to the N2NQ, so that only specific data is being sent to the receivers. Responses, on the other hand, are used as a confirmation for the N2NQ, to make sure that each message has truly been received at the receiver. It would be possible to develop a feature, where the N2NQ instance could resend messages, if responses indicated that errors were found. This, however, has not been implemented for the current prototype, as no instances of message corruption could be noticed during testing. Both requests and responses can be separately turned on or off by the user.

4.2.3 N2NQ

The N2NQ is the core of the prototype and the central element in the message distribution functionality. The main purpose of the N2NQ is to receive messages from the senders and store their contents into a queue. After this, the messages are retrieved from the queue and sent forward to the correct receivers. The N2NQ instances were designed to operate as a single process from the beginning, although, it also utilized multithreading for connecting to the different external senders and receivers.

The internal operations of N2NQ are composed of three distinct parts: receivers, senders and the queue handler. On a basic level, these internal receivers and senders work much in the same way as their external counterparts. The internal receivers collect the messages sent from the external senders and store them to the central queue. The internal senders then pass on the queued data to the external receivers. The set of internal senders and receivers utilize multithreading within the same N2NQ process and their quantity is directly influenced by the amount of corresponding external senders and receivers implemented. While on surface nearly identical, the internal and external services also

have unique tasks and operations. The key difference is the way the internal services communicate with the dedicated queue in the core of the N2NQ process. The receivers transfer new messages into the queue, while the senders read through the queue and transfer messages forward to the correct receivers. Based on the instructions given, the internal senders are able to evaluate each message and whether it should be sent forward.

The queue itself is effectively a one-way linked list, with new items being added from one end and removed from the other. While the internal receivers are tasked with the introduction of new messages, management of the queue is otherwise being handled by the queue handler, which main task revolves around the removal of unnecessary elements. The way messages are being managed in the queue can be performed in two distinct ways. These different queue modes either remove messages after a set amount of time or immediately after being sent. When the former mode is in effect, each message is being received by all of the implemented receivers, while in the latter, all of the messages get sent only once to a random receiver. The first queue mode could be seen as transforming the prototype into a broadcaster, while the second one operates more like a load balancer. For simplification, these two modes will be referred to by the names of these use cases, in the rest of this thesis. The functionality provided by these modes can be further enhanced with the use of requests sent by the receivers. The use of either queue mode is controlled by the user.

4.2.4 Inter-process communication

Since the three prototype elements each run in different processes, a stable method of inter-process communication was required. The established communication works with data messages first being sent from the sender processes to the N2NQ, and from there to the receiving processes. Each of the messages are sent via TCP data streams, with connections that have to be established before the sending can begin. While the communication has been designed, in the first hand, to work with messages going directionally to one-way only, two-directional messaging can be also used to achieve specific functions. This was relatively straightforward to include due to the nature of TCP.

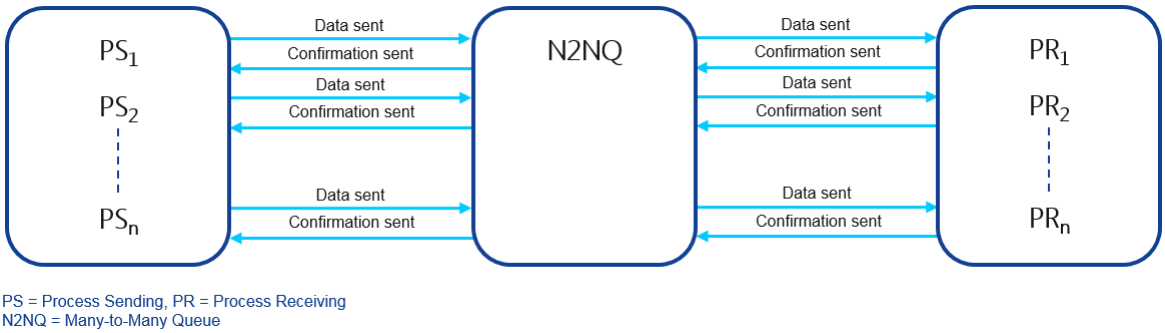

Figure 8. A model of communication in N2NQ.

Figure 8 displays an overall model of communication for the prototype, where messages can also be sent back from the receiving processes to the N2NQ. This way the program is able to confirm that the data has been successfully transferred to the receiving end. This response feature can be freely turned on and off when necessary, providing possibilities to run the prototype with varying settings. This figure also aims to showcase the possibility of using a flexible number of sending and receiving processes, both of which are initialized as singular group processes, with multithreading for each individual sender and receiver, based on the quantity of microservices required.

Going forward, the prototype was also expanded with the implementation of the aforementioned request mechanic, in which the data messages are only sent forward from the N2NQ instance, after being requested by the receiving processes. When combined with the response messages, the runtime becomes significantly more controlled by the receiving processes. The basic communication model, with both of these features, can be visually observed from the sequence diagram found in Figure 9 below. As with the confirmation responses, the requests are also completely optional and can be turned on and off easily. Additionally, the requests also include a feature, that allow for messages to be guided to specific receivers, based on their individual senders and message lengths.

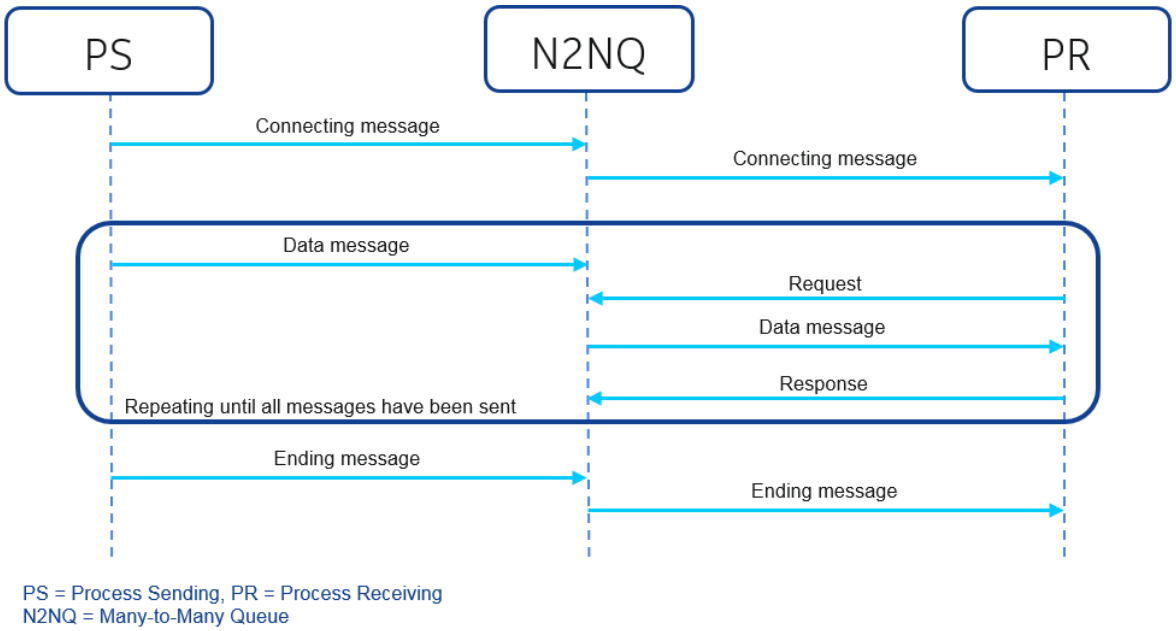

Figure 9. A sequence diagram for communication between N2NQ prototype processes.

As can be viewed in the figure, there are also specific connecting and ending messages present which, unlike the requests and responses, cannot be turned off. These messages are used every time at the beginning, when the runtime is started, to confirm that the processes have successfully connected, and at the end, to signal that the prototype run has reached its end and the TCP connections can be terminated. The direction of these connecting messages is always from senders to the receivers, which in practice translates to messages going from the external senders, found in the sending process, to the N2NQ's internal receivers, and from the internal senders to the external receivers, which are found in the receiving process.

While this basic communication structure remained largely the same throughout the development of the entire prototype, this only depicts functionality running on a single VM. Since the prototype was designed to mainly operate in a cloud environment with multiple VMs, a version like this had to also be implemented. The cloudification of the prototype meant, that each of the prototype processes, senders, receivers and the N2NQ, had to be placed in different VMs, located in the same cloud. In addition to this, the option of running the senders and receivers from the same machine as the N2NQ instance was also retained, so that the differences between a local and distributed version could be studied. Going from a strictly local perspective to a cloud required fairly small changes to

the communication methodology on a code level, although the messages had to be routed to different addresses.

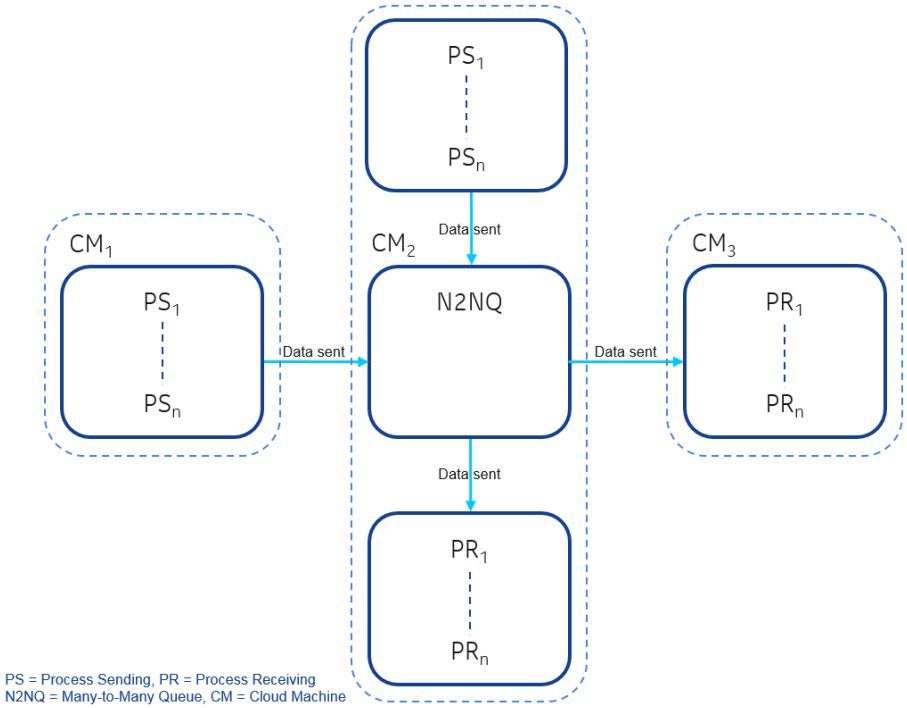

Figure 10. A simplified communication model of the cloud version of the prototype.

A simplified model of this cloud version can be seen in Figure 10. While not necessarily apparent in this figure, all of the feature and message directions, that were presented in Figure 9, are also available in this version. There are just multiple alternative placements for the processes to be found. It was also speculated, that the prototype could be expanded into a multi-cloud communication setup, although this was left out of the scope of the thesis. Eventually, a GUI was also built for the prototype, for additional convenience in performing tests.

4.2.5 GUI

After weighing a multitude of options, the GUI for the prototype was chosen to be built using Qt, a cross-platform application framework, that is able to extend on the core features offered by the C++ language. Qt offered a comprehensive set of tools specifically for development of graphical C++ software, as well as detailed documentation and a variety of suitable guides for beginners. At this point, the development was also moved

from NetBeans to Qt Creator 5.9.1, which was a similar IDE that offered more direct support for GUI design and development. [28]

For the usability of the prototype, it was important that all of the core processes, the N2NQ, the receivers and the senders, could be started from the GUI and in any order, with a press of a button. The number of senders and receivers would also need to be easily controllable, so that a variety of test cases could be implemented for the testing of the prototype. The formation of the GUI began with an implementation of these core features. Since retaining low-latency was a key goal. it was clear that the GUI could not be allowed to result in any unnecessary stress for the N2NQ. For this reason, it was important that there would be no direct involvement between the GUI and the N2NQ instance, at least during runtime.

The aim of the designed GUI was to completely separate the N2NQ instance from the GUI, as well as avoid all other unnecessary sources of stress on the queue. This meant that the source of the GUI information had to be the sending and receiving processes themselves, instead of the N2NQ instance. One of the reasons why the sending and receiving processes were designed to operate as singular processes, with multithreading, was to facilitate and simplify this functionality.

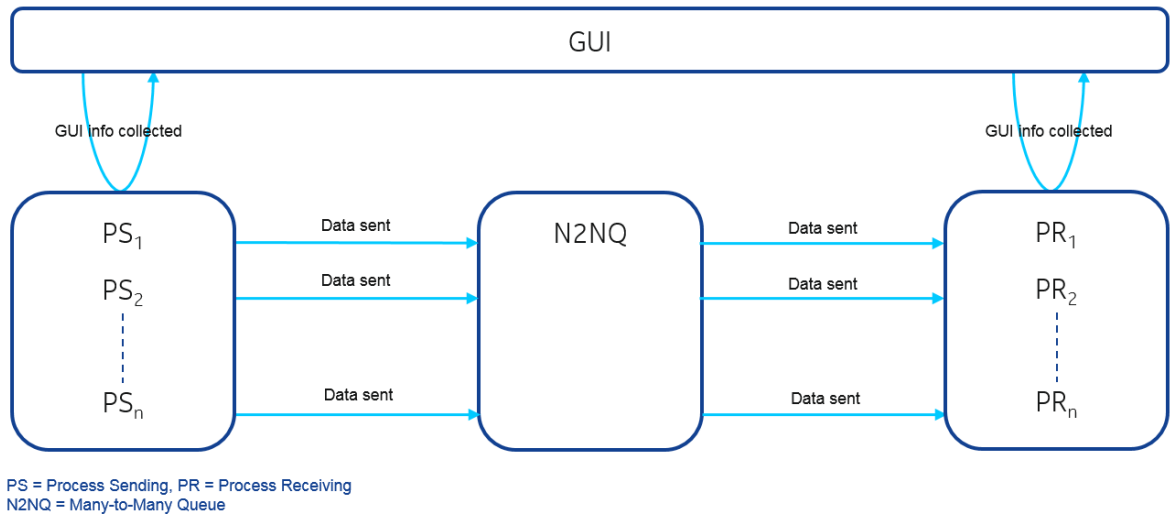

Figure 11. Communication model between the GUI and the prototype.

As can be seen on Figure 11 above, the sending and receiving processes fully handle communication with the GUI, leaving the N2NQ free to process as fast as possible. While each of the core prototype processes could be located in the cloud, each in a VM of their own, the GUI itself was operated from a local VM and communicated with the senders and receivers from there using the Secure Shell (SSH) protocol. The only direct connection to the N2NQ it had, was the ability to launch the instance with the correct starting settings.

The finished version of the GUI can be seen in Figure 12 below. In this version, all of the presented features can be toggled to the users liking. The left side of the GUI is designed to serve sender related functionalities while the right side does the same with receivers. Under the "Sender amount" toggle, information can be found on all of the implemented senders. This includes the individual values for delay, upload rate and packet counter. Above them, an aggregate of the individual values can be found. Similar functionality is found on the right side for the receivers, where the delay has been replaced with latency. To the left of each senders and receiver, a button is located, which allows to user to open a text file, containing all of the data concerning that specific microservice, recorded during the latest performed test run.

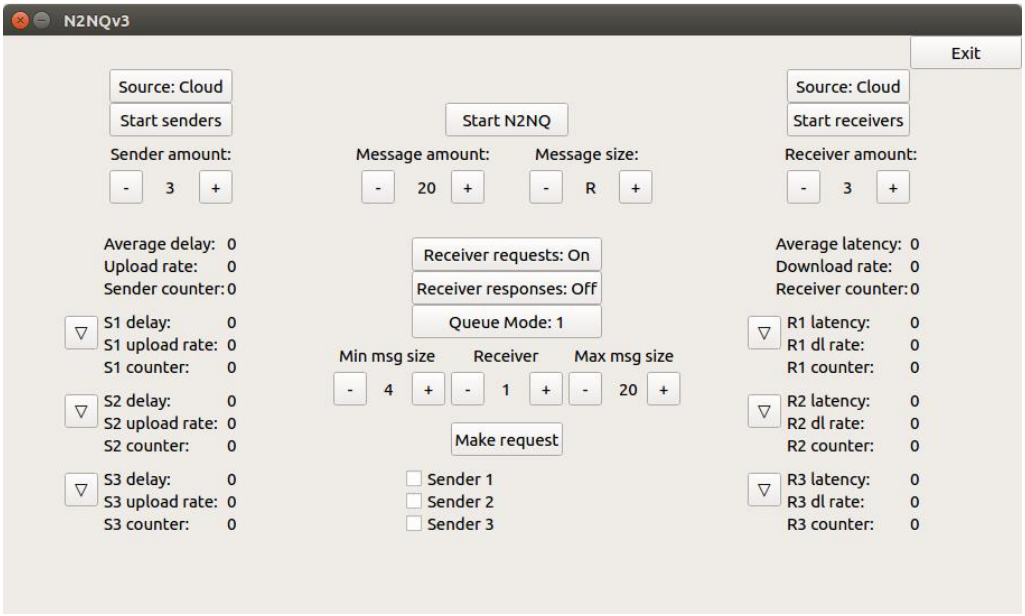

Figure 12. GUI of the prototype.

In between the senders' and receivers' functions, all of the additional functionalities are located. These include the ability to change the queue mode being used, which can be toggled with the "Queue mode" button. In the case of the GUI, Queue mode 1 represents the broadcaster mode and Queue mode 2 represents the load balancer. Below the "Queue mode" button, the user can find all the settings governing the use of responses and requests.

As mentioned previously, the possibility to use responses was made into an optional feature, which can be activated from the "Receiver responses" button. This feature was then expanded into an additional ability to have the receivers to be able to send unique requests to the N2NQ, which could be used to define what kind of messages they wanted to receive. Each of the settings shown below the "Queue mode" button, are only present when the requests have been turned on with the "Receiver requests" button. These setting allow the user to send commands directly to the receivers, which in turn make them forward a request to the N2NQ about what kind of messages they should receive. The settings allow for the user to select precisely which senders' messages should go to which receiver, as well as which sized messages the receivers should get. These requests can be initiated before or during runtime and can be sent with the "Make request" button.

As the prototype had to work in a cloud environment, this also necessitated some additions to the GUI, in order to benefit from possibilities offered by this environment. Since it was decided, that the senders and receivers could both be operated on a VM of their own, or in the same one as the N2NQ, buttons were added to the GUI to enable this functionality. Above the "Start senders" and "Start receivers" buttons, new "Source" buttons were added that could be used to toggle the location of both processes. This version of the GUI is also the one, that was used to perform final testing and produced the results that are presented in chapter 5.

5 TESTING AND DISCUSSION

This chapter focuses on testing in three steps. First the used methodology and test cases are presented. After this the results of these cases are showcased graphically and explained in detail. Finally, the chapter concludes with an evaluation and discussion of the significance that these results hold.

5.1 Testing

As mentioned before, the most important key measurement chosen for the thesis was latency. This was important to measure in two ways, on average and for every message received. The main goal was to measure which circumstances would yield the most beneficial results for the sake of latency and whether there were any breaking points to be found for the prototype, as well as where these would be reached.

5.1.1 Testing methodology

In practice, the latency measurements were performed by checking and saving the precise time when the message was created and before it was sent, then repeating the process when the message was received at the other end and comparing the two figures. Send times for messages were included in the message header for each message. When messages reached a receiver, the time was measured again and the difference was calculated. Each of these latencies were then automatically saved into specific text files, corresponding with the receivers that measured them. At the end of each run, this collection of data files was combined into a single master file that could be studied easier. The prototype also calculated the combined average latency of all of the messages in real time. This average latency could then be seen directly from the GUI.

Each of the designed tests were comprised of a predetermined set of test runs, each with unique prototype settings. All of these test runs produced an overall average value of latency, that was calculated based on the messages that were transmitted during that run. This average value of latency was calculated using only 95^{th} percentile results from each test run, and everything above that was discarded. Averages were chosen as the

utilized measurement, as they provided a straightforward method for comparing each test run to each other. The reasons for choosing to utilize the 95th percentile results are presented in the "Results" subchapter. With these measurements, the average changes could be observed between each test run. These results were then transformed into a set of graphs, providing a more approachable method for study. In addition to average measurements, graphs were also produced from percentile calculations, based on individual latencies measured during specific test runs. These were meant to highlight the variations and extremities of latencies found within singular test runs.

While operating on a local level, performing latency calculations was straightforward, but moving to a distributed cloud environment posed possible issues that had to be solved. Since each of the cloud machines were separate entities, it was important that there would not be any differences in their internal clocks. So that the measurements would be accurate, all of the cloud machines were synchronized with each other using the Network Time Protocol (NTP). This way the measurements could be performed in the same manner as with local runs, without compromising accuracy. [29]

5.1.2 Test cases

The testing was performed with seven unique cases, each designed to verify specific aspects of the prototype. Many of these tests were also run with multiple different test sets, producing a broader set of test data. A summary of the used settings for each test can be found on Table 1 below.

Table 1. Used settings for utilized test cases.

Test case:	Test group amount:	Message size:	Message amount:	Delay:	Sender & receiver location:	Requests or responses:	Queue mode:
1	1	**Variable**	50	1 s	External	Both off	Broadcaster
2	1	50 B	**Variable**	1 s	External	Both off	Broadcaster
3	4	50 B	50	**Variable**	External & internal	Both off	Both tested
4	2	50 B	50	0.5 s	External	Both off	Both tested
5	2	50 B	50	0.5 s	**Variable**	Both off	Both tested
6	2	50 B	50	0.5 s	External	**Variable**	Both tested
7	2	50 B	50	0.5 s	External & internal	Requests on	**Broadcaster & load balancer with requests**

When viewing the table above, a specific bolden setting can be found in most of the test cases. This is meant to highlight the specific key setting for those tests. The aim of these test cases is to measure the impact of this specific setting. Test cases 4, however, is an exception in this table and its key setting is explained in the description below. A more precise rundown for each of the test cases is presented next:

Test case 1: The aim of test 1 was to see whether changing the message size would introduce noticeable impact on latency. The test was executed in six sets, each with different message sizes. These sizes were: 0.1 kB, 1 kB, 2 kB, 4 kB, 8 kB and 16 kB. Each of these sets were then run with increasing quantities of senders and receivers, to see whether the significance of message size increased with additional operational microservices.

Test case 2: Test 2 was designed to verify what the significance of message amount was on overall latency for the message transfer. This test was performed in five sets with increasing amounts of messages for each sender. These amounts were: 1, 25, 50, 75 and

100 messages. Each of these sets were then run with increasing amounts of senders and receivers, from one, all the way to a hundred.

Test case 3: The goal of test 3, was to see how impactful the length of delay between every message sent, or the rate of traffic, was on latency. This became the most thorough test performed, including four distinct test group, based on whether the senders and receivers were internal or external as well as the queue mode being used. Each of these group settings were then tested with a set of ten different lengths for delay, from 1 to 0.1 seconds. Finally, each delay was run on a variety of sender and receiver quantities, each larger than the previous.

Test case 4: The goal of test 4 was to gain a more precise picture on how much increasing the quantity of senders or receivers impacts latency, and how they differed from each other as causes of latency. This test was divided into two groups, one performed with the broadcaster mode and the other with the load balancer. Both groups were then split into three sets, based on the specific service that's quantity was being increased. In one of the sets only the number of senders was increased, the second one focused on only adding more receivers, and the final group had both of them increase. The impact caused on overall latency was then measured with multiple increasing levels of quantities.

Test case 5: Test 5 aimed to showcase what effects the location of the senders and receivers had on latency of messages. This meant measuring whether latency increases when senders and receivers were running on the same VM as the N2NQ, or on a different one. To get a more comprehensive picture, the test was performed in two groups for both queue modes. Both groups were then executed with four sets, based on the location of the senders and receivers. These groups were: both senders and receivers as internal, only senders as external, only receivers as external, and both senders and receivers as external. The number of both types of microservices were then steadily increased as each set was executed.

Test case 6: Test 6 focused on the aspect of requests and responses, and whether turning them on hinders latency for message delivery. The test was performed in two groups, for

both developed queue modes. Both queue modes were then run with four distinct test sets, each of them having a different combination of requests and responses settings. These test set settings were: requests and responses turned off, only requests turned on, only responses turned on, and both requests and responses turned on. Each of these settings were then tested with an increasing number of senders and receivers.

Test case 7: The seventh and final test focused on comparing two different methods of achieving the result, where each message was sent to only one receiver. These were the regular use of load balancer mode, and the utilization of broadcaster mode with specific requests, that forced each sender to only transmit to a specific receiver. Both of these situations were tested in two test groups, with either external or internal senders and receivers. All variations were then tested with increasing quantities of senders and receivers.

5.2 Results

Each of the designed test cases were performed on their own, one at a time. As the test results were first observed, notable spikes of latency could be noticed in some of them. These spikes were not very common, but occasionally so significant that they could skew the average latencies by a significant margin. This is clearly observable in the Figure 13 below. The figure was formed based on a single test 5 test run, where the senders were set as internal, receivers as external and the quantity of both was 50, with load balancer as the chosen queue mode.

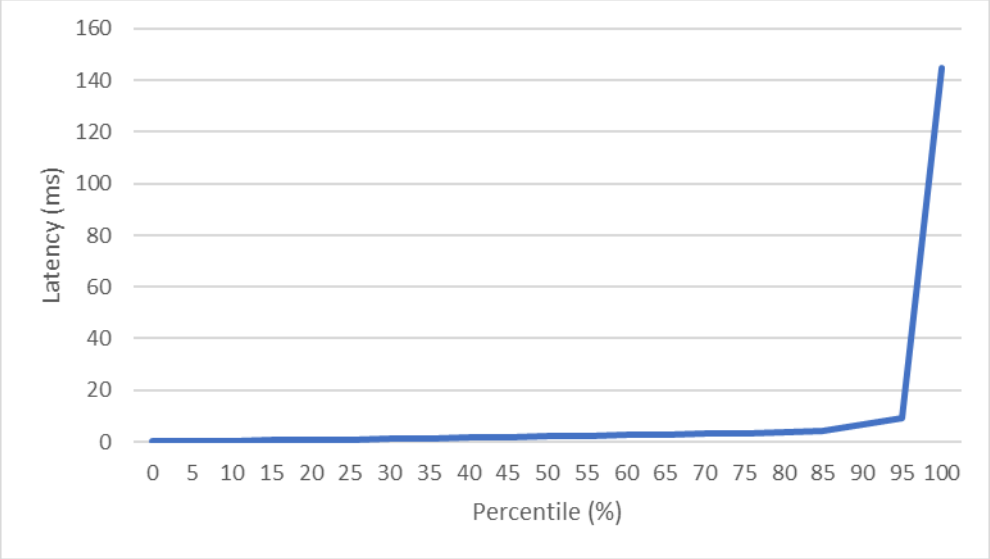

Figure 13. Latency by percentile, from a single test run.

In the presented test run, results in the highest fifth percentile dramatically shoot up compared to the rest of the 95th percentile. Even with a large amount of measurements, 2500 in the case of this test run, spikes this high can impact the average calculations in a noticeable way. While this only showcases the issue during a single run, this was not an isolated case and similar spikes were observed during other tests as well. Due to this, it was decided that the utilized average latencies would be based on the 95th percentile measurements. This way the averages could be presented in a more stable manner. The spikes themselves were, however, studied as well as a separate topic.

Based on these 95th percentile results, a set of graphs were created to visualize the results from each performed test set. These key findings for each test are presented below, in a numerical order:

5.2.1 Test case 1

The first test provided a set of results on the effect of message size on average latency, experienced by each packet passing through the prototype. While the results mainly coincided with the assumption that increased message sizes would also lead to increases in latency, there was also a notable exception to be found. For some reason, when using a message size of 4 kB, the measured latencies tended to become equal or higher compared

the measurements performed with the larger 8 kB messages. With 50 senders and receivers implemented, the impact became even more notable, rising even above the results gathered from the 16 kB test run, which was the largest one performed. As can be observed from Figure 14 below, the abnormal results of the 4 kB tests show a clear exception when compared to the other graphs.

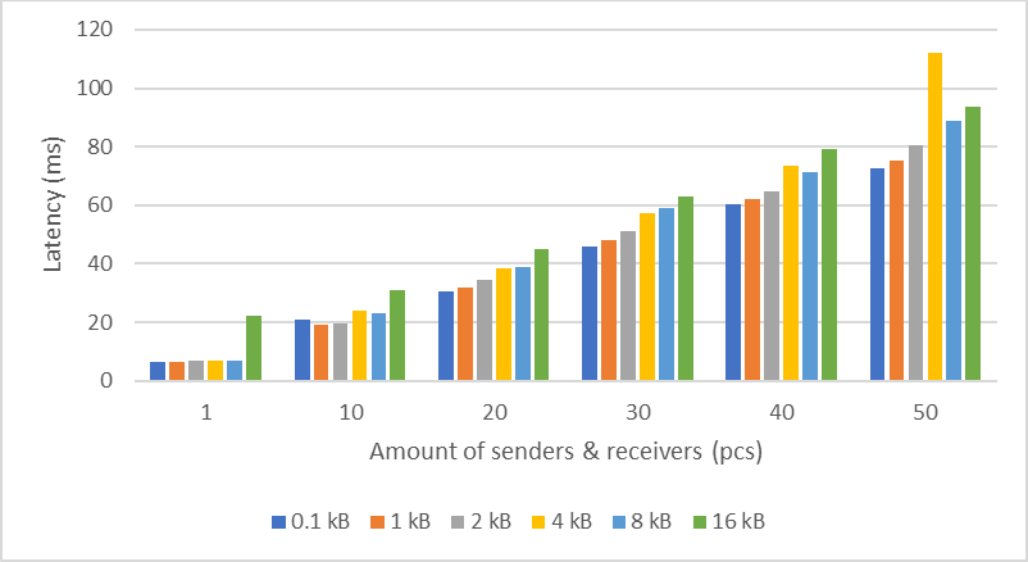

Figure 14. Impact of message size.

What made these 4 kB messages behave different from the others is not clear, but the rest of the results showed more predictable behavior. With each increase in message size, a steady rise in overall latency was notice throughout the tests. Regardless, it was decided that the rest of the tests would all use a standard small message size of 50 B, so that none of them would be impacting by this setting. This way rest of the tests could be executed based on their specific chosen key settings with consistency.

5.2.2 Test case 2

Test 2 was used to see, whether increasing the amount of messages being sent would impact latency. The main assumption, prior to testing, was that the increasing message amounts would not, on their own impact latency. Rather, it was assumed that having more messages being sent, could lead to a more stable set of results, simple because the amount of raw data would be increasing. The received results, however, did not show a significant

rise in stability nor did they visualize any significant impacts on the produced average latency levels. This can clearly be seen in Figure 15 below.

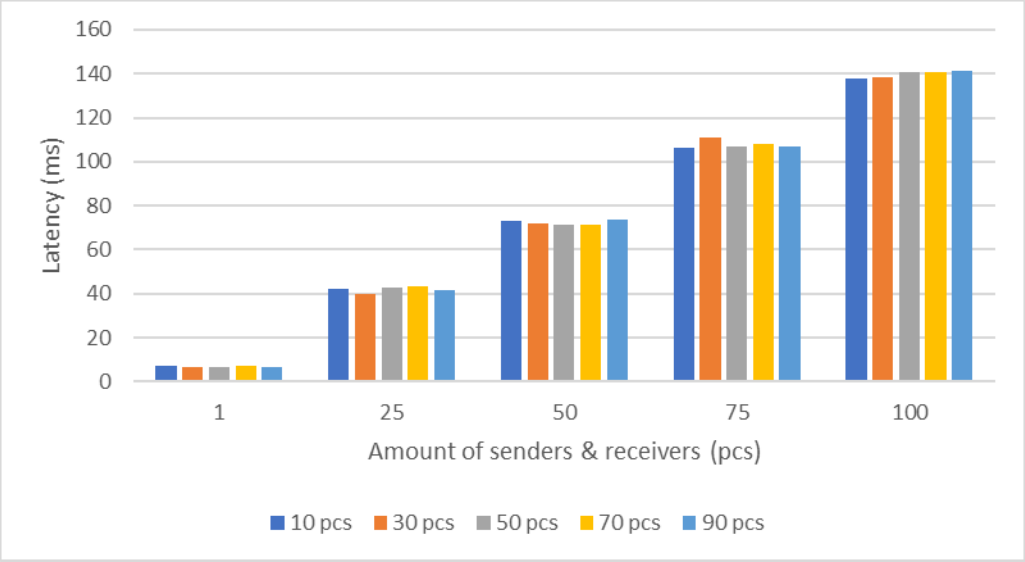

Figure 15. Impact of message amount.

While the stabilizing effect of increased message amount should still apply in certain cases, when abnormalities are encountered, this does not seem to be a significant effect for most test runs. The same effect could also be achieved multiplying the number of runs performed. Regardless, the case by case latency for each message does not seem to be affected by this setting. Based on this result, the message amount was locked as a compromising 50 per sender, for each of the following tests.

5.2.3 Test case 3

As mentioned previously, the third test was the most comprehensive of the ones performed. As such, it also produces a variety of results. It was evident that decreasing the delay between sent messages had a significant effect on prototype traffic. This resulted in visible increases in latency, at least when broadcaster mode being was utilized. As can be observed from Figure 16, with external senders and receivers being used, latencies seemed to remain stable, regardless of the delay used, until the 50 sender and receiver amount was reached. After this point, decreasing the delay started to have a noticeable impact on the test runs with the highest traffic rates.

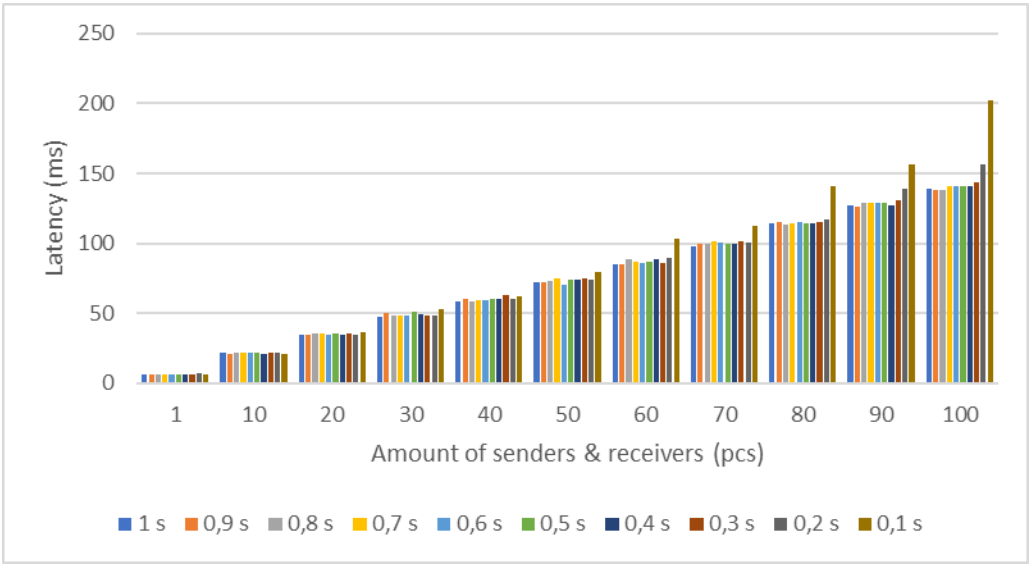

Figure 16. Impact of delay length, in broadcaster mode with external microservices.

As the quantity of senders and receivers increased, test runs with delays in the range of 0.2 to 0.1 seconds started to produce noticeably higher numbers in terms of average latency. It seems evident, that at this point the prototype has reached its limit and cannot keep up with the rate of traffic any more. This impact becomes even more self-evident when the implemented microservices are turned from external to internal, as Figure 17 clearly shows.

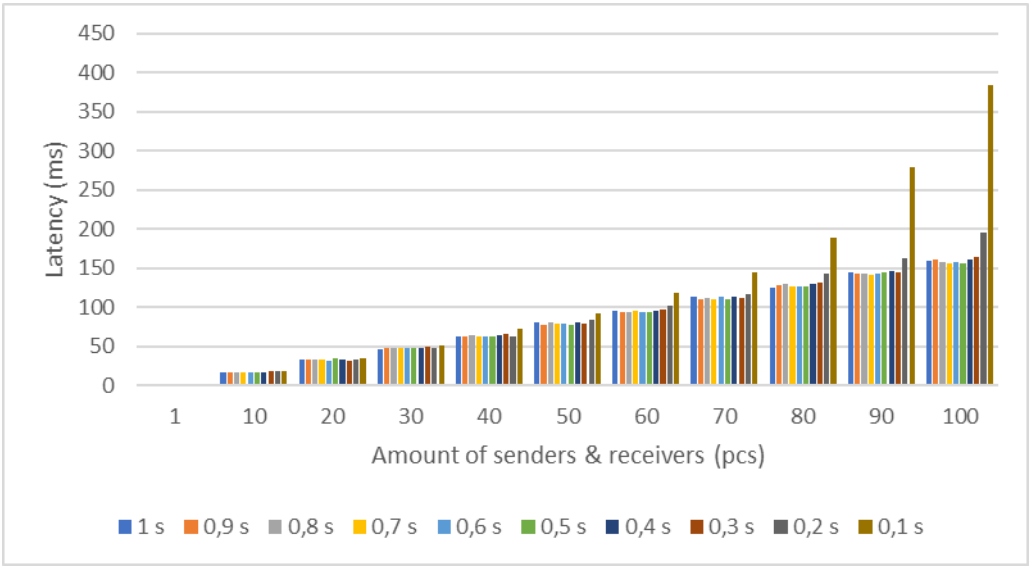

Figure 17. Impact of delay length, in broadcaster mode with internal microservices.

While the state of uncontrollable traffic seems to occur at the same point, 50 senders and receivers, as with external microservices, the severity of dramatic latency rise is greatly increased with internal microservices. Since each of the prototype processes, senders, receivers and the N2NQ itself, are running in the same VM, the stress incurred is also significantly larger. These kind of results, however, do not recur when the same tests are executed with the load balancer mode being utilized.

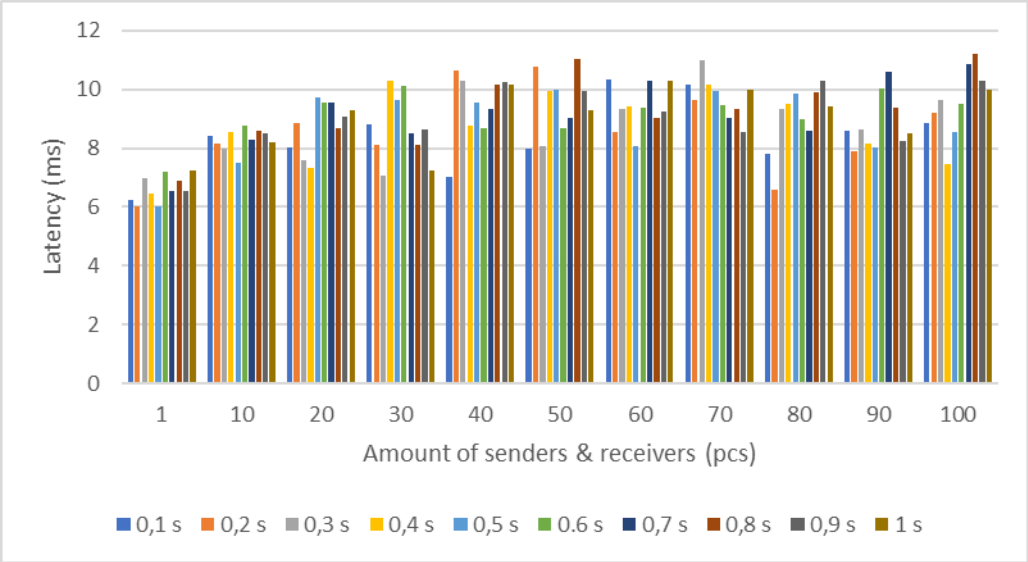

Figure 18. Impact of delay length, in load balancer mode with external microservices.

As Figure 18 shows, the average latency numbers remain relatively consistent when the load balancer mode is utilized, even when microservice quantities are increased and the delay between sent messages is decreased. With external microservices, the average latency hovers fairly consistently between 6 and 10 milliseconds, regardless of the increasing traffic caused by the other factors. The average latency for the entire test set measured as 8.8 milliseconds. Even when differences are observed, they are relatively small and seem to be occurring randomly, regardless of changing prototype settings. This consistency is even more apparent when the microservices are being run internally.

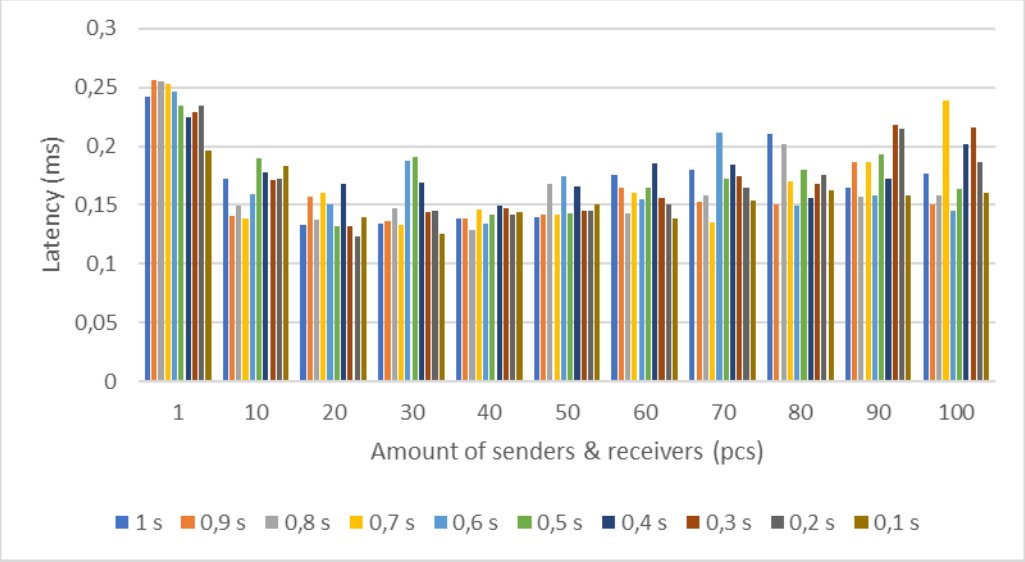

Figure 19. Impact of delay length, in load balancer mode with internal microservices.

As can be viewed from Figure 19, with senders, receivers and the N2NQ all running in the same machine, the average latencies become even more stable, and significantly lower with the average, for the whole set, being just 0.17 milliseconds. Since the overall latency remains so low in all of the runs, even a small spike can make the graph jump in a noticeable way. However, even these cases are caused by a relatively low amount of hang ups and most extreme example still leaves the average latency well below 0.3 milliseconds. Based on the test 3 results, a decision was made to limit the delay to a safe 0.5 seconds for most of the following tests. This allowed for further test cases to be performed with less time, without compromising the validity of the tests on either queue mode.

5.2.4 Test case 4

The fourth test aimed to give more insight on what kind of effects would occur from increasing the number of senders and receivers. This led to some of the most unexpected results in the thesis. As Figure 20 below shows, when using broadcaster mode, the effect of increasing senders did little to increase latency. Compared to this, implementing additional receivers produces a completely opposite effect, making the latency increase in a nearly linear fashion. When the senders were increased at the same time, an increase was also noted, but on an average of 6.3 milliseconds higher level, throughout the test runs. Running the same tests with the load balancer mode, however, produced more unusual results.

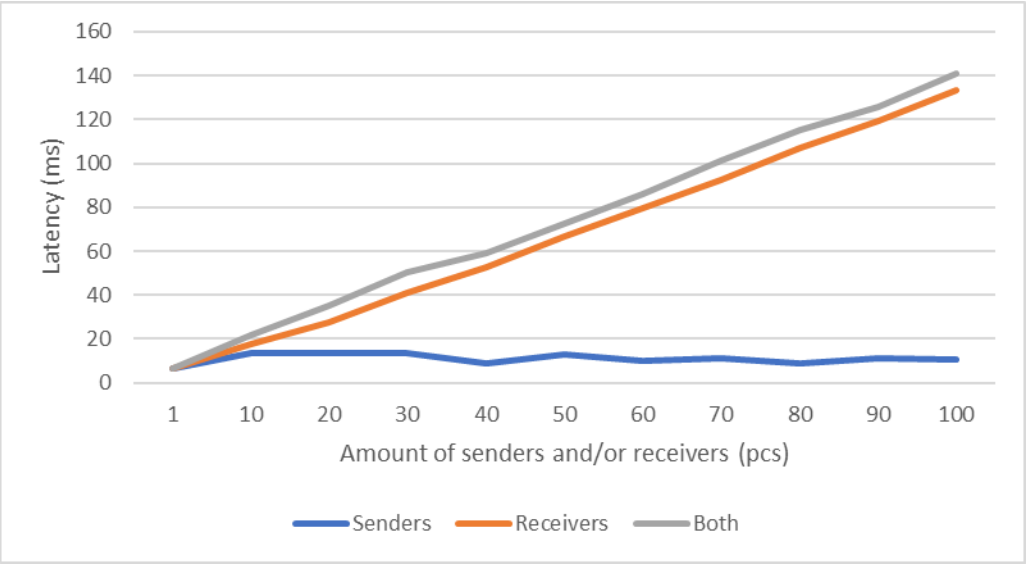

Figure 20. Impact of microservice amounts, in broadcaster mode.

With the load balancer mode, it was expected that all of the test groups would produce similar results, regardless of which microservices were increased in numbers. This also seemed to be the case, until the amount of services exceeded 80. As can be seen from Figure 21, at this point, when only the amount of receivers was increased, the average latency started to rise significantly. The graph, however, remains on a similar steady level, when only senders are increased or when both microservices are increased.

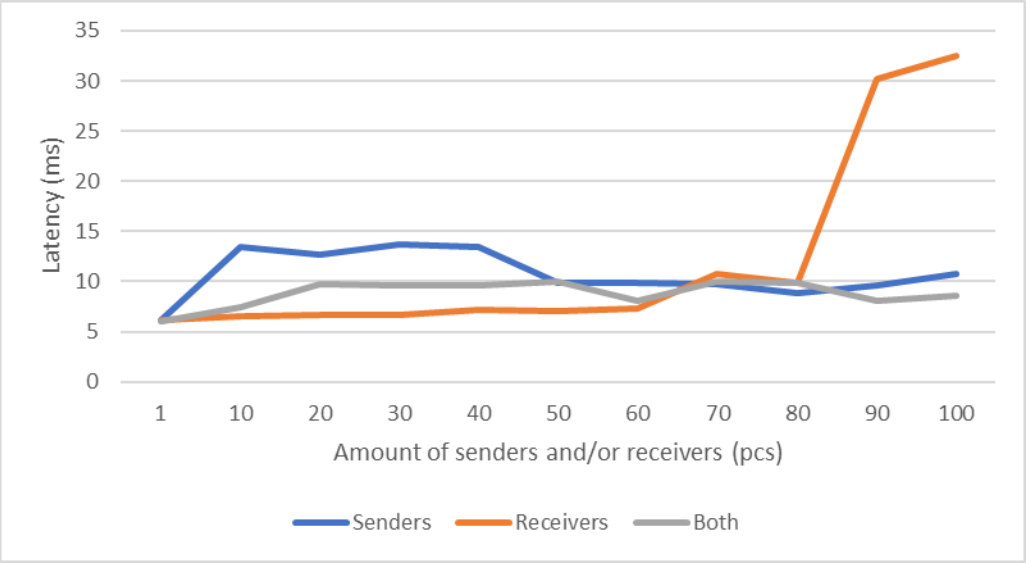

Figure 21. Impact of microservice amounts, in load balancer mode.

The specific reason for the latency rising, when only receiver amount is increased, did not prove to be self-evident. Regardless, both microservice amounts were increased simultaneously during the following tests, so that similar issues could be avoided.

5.2.5 Test case 5

The fifth case also provided interesting results on the nature of the prototype. The test was executed in two sets, for both queue modes. During the third test, it could already be noticed from the results, that when microservice numbers were kept small, internal operations run on a clearly lower latency on both queue modes. This result can be observed even more evidently with the results from this fifth test.

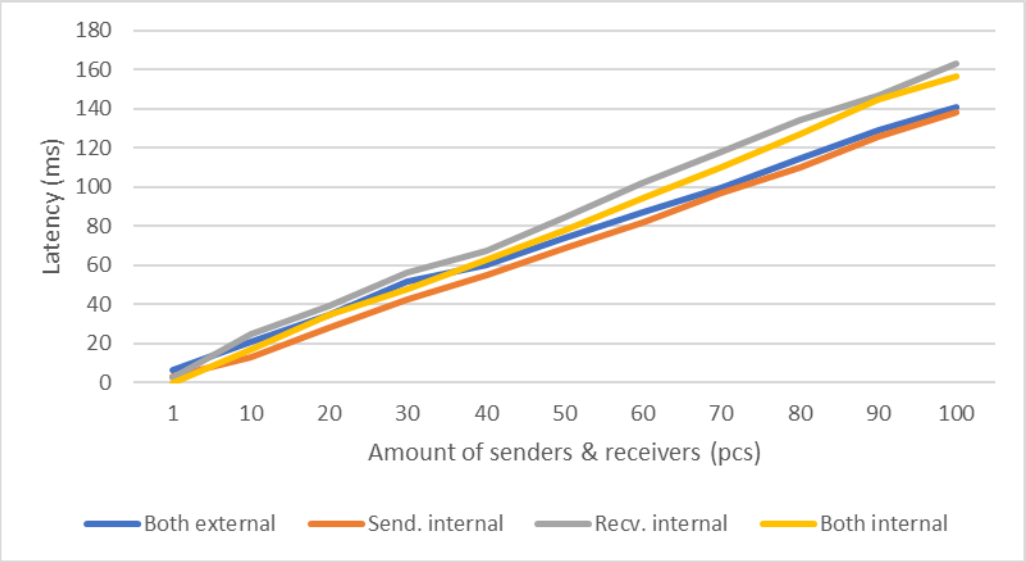

Figure 22. Impact of microservice location, in broadcaster mode.

As Figure 22 above shows, situations where receivers were made internal, managed to provide better latency results up to 10 concurrent microservices. At this point, however, internal measurements begin to fall behind and steadily rise faster than those cases, where the receivers were made external. On the other hand, the placement of senders does not seem especially impactful in any of the tests. With the load balancer mode, the results became quite different.

As was evident with the graphs based on test 3, when load balancer mode has been initialized, increasing the quantity of senders or receivers does not impact latency significantly. This can also be seen clearly on Figure 23 below. What does impact latency, however, is whether they are being run externally or within the same VM. When using fully internal microservices, load balancer mode really shines on latency.

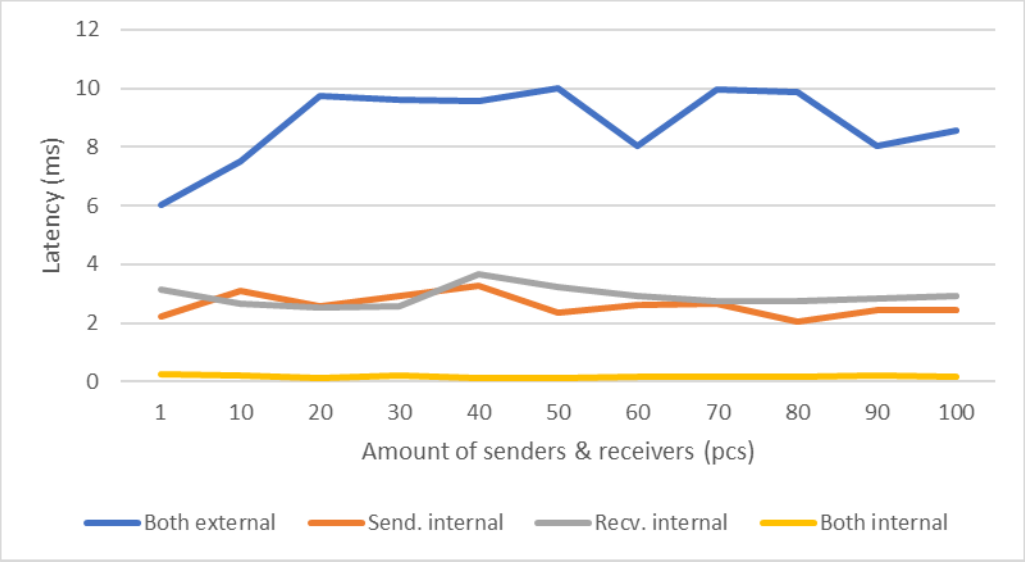

Figure 23. Impact of microservice location, in load balancer mode.

With only one type of the microservices, senders or receivers, set as internal, the benefits are less evident as the average latencies rise to multiple times higher levels. It is still interesting to see, that receivers did not prove to be any more taxing for the prototype than the senders. Naturally, the highest latencies are seen when both services have been set as external, rising to over twice as high levels than if only one of them is external. Regardless, the latencies remain constantly lower than in the case of the broadcaster mode.

5.2.6 Test case 6

With test 6, the impact of requests and responses was valued. These test sets provided results, with a large degree of variation. This was especially noteworthy with the test runs where both requests and responses were turned on. Due to this, the results from these specific runs were placed on a separate figure of their own.

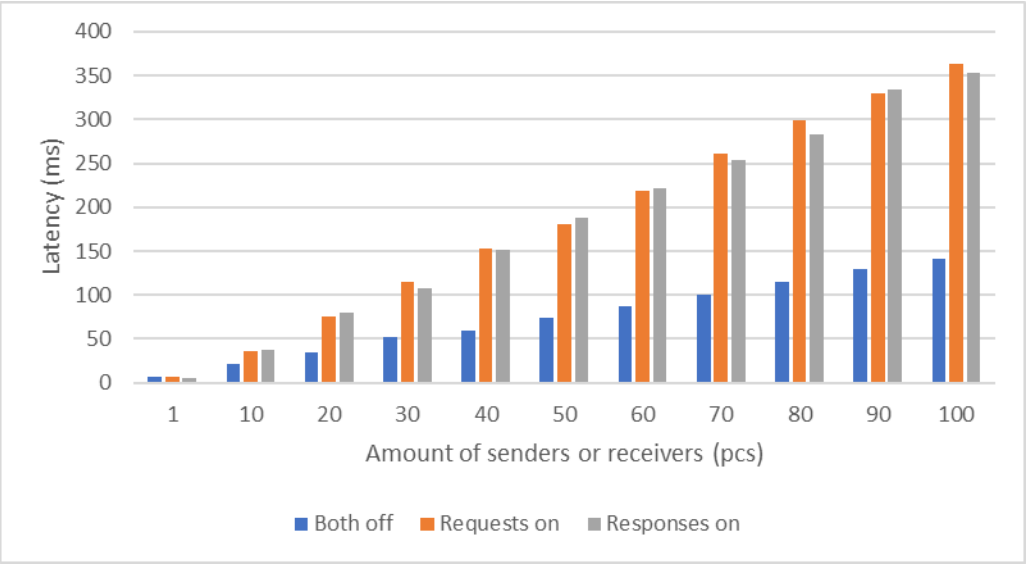

Figure 24. Impact of requests or responses, in broadcaster mode.

As Figure 24 shows, both requests and responses increased average latency by a significant margin. While the graph steadily grows even with these turned off, either of them being used caused the results to more than double in most instances, at least with the broadcaster mode. As can be observed, this rise is fairly linear and no notable breaking points were noticed, although it is possible that decreasing the delay could lead to this being the case.

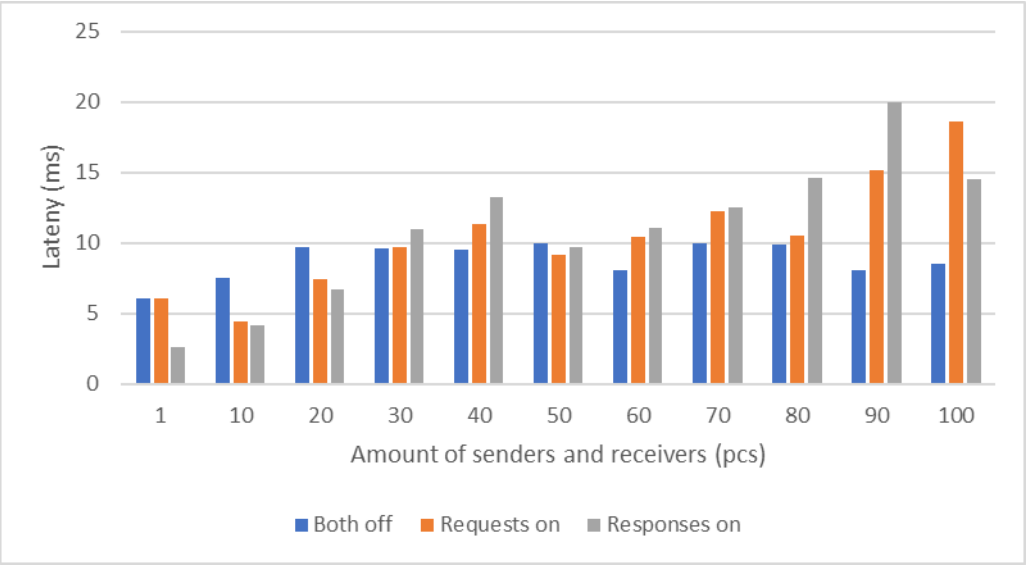

Figure 25. Impact of requests or responses, in load balancer mode.

The results from testing with load balancer mode offered a much more unstable set of figures, compared to the ones in the previous figure. As can be seen from Figure 25, there seems to be a slight increase in the latencies when requests or responses are being used, especially when sender and receiver numbers are being increased. These results, however do not indicate a similar kind of steady growth as many of the previous graphs. Regardless, this did not remain as the only unexpected graph based on these test sets, as the following figure, where both requests and responses have been utilized, shows.

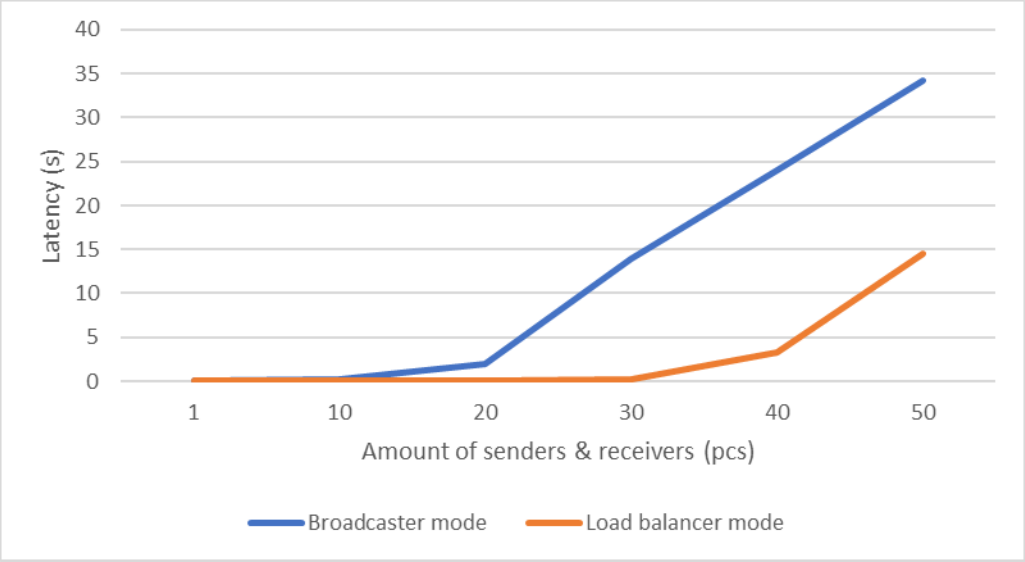

Figure 26. Impact of requests and responses, in both queue modes.

The most fascinating result from this test came apparent, when studying the results gotten from running the prototype with both requests and responses, regardless of queue mode. Even though both tests were performed with external senders and receivers, as well as a steady rate of traffic across the test runs, similar issues arose as in test 3, where too high of a traffic caused the prototype to not be able to keep up. As can be seen in Figure 26, in the case of both queue modes, there is a clear cutoff point, after which the latency starts to rise uncontrollably. In the case of broadcaster mode, this point comes after 20 microservices have been reached, where as in the case of load balancer mode, a similar point can be witnessed at the 30-microservice mark.

5.2.7 Test case 7

The final test measured whether the use of load balancer mode or requests would produce better results, when aiming to get each message to be sent only once to a single receiver. This test was performed separately with both external and internal senders and receivers. While the overall results indicate, that the use of the load balancer mode resulted in significantly lower latencies, there were also noticeable differences between test runs with internal and external microservices.

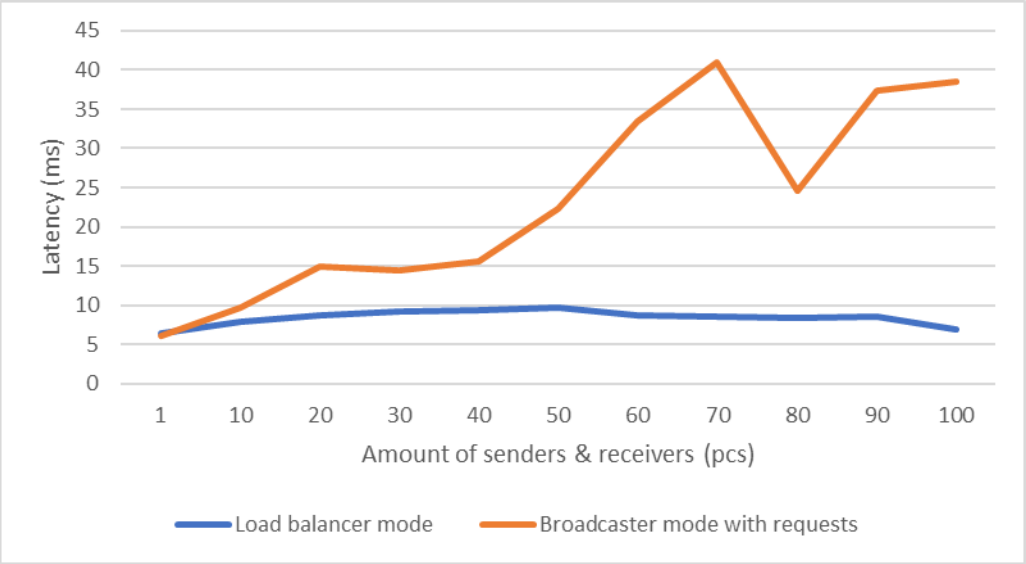

Figure 27. Load balancer mode vs. broadcaster mode, with external microservices.

When external microservices were employed, the use of the load balancer mode seemed to produce a consistent result, without much deviation, regardless of the service amounts, which was to be expected, based on previous tests. The use of broadcaster mode with requests, however, produced a significantly more uneven set of results. As can be seen in Figure 27, the requests graph starts to rise relatively steadily until the quantity of 20 microservices has been reached. However, after this point the measurements begin to show inconsistencies.

The graph seems to first even out on a 15-millisecond level, between 20 to 40 microservices. After this, a sharp rise in latency can be notices, followed by notable spikes of latency rising and lowering between the ranges of 25 and 40 milliseconds. These spikes

are surprisingly significant and the reason for their appearance did not seem obvious. What makes these results even more fascinating, are the oppositely consistent results gathered when internal microservices were used.

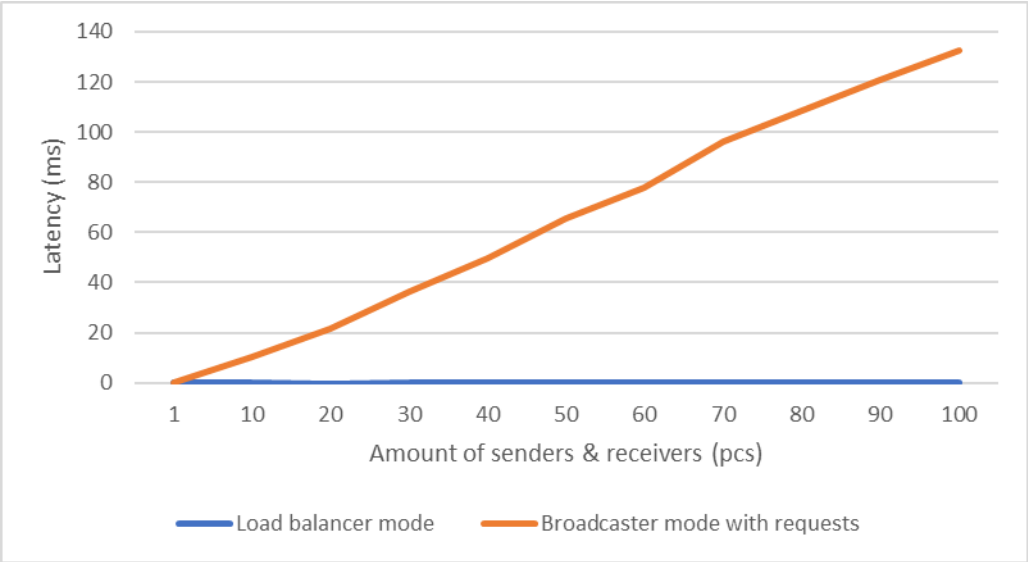

Figure 28. Load balancer mode vs. broadcaster mode, with internal microservices.

In the case of the test set with internal microservices, the executed load balancer mode runs once again produced familiar results with exceedingly low latencies and no notable variations, even when the service amounts were increased, as can bee observed from Figure 28. On the other hand, the test runs with the broadcaster mode and requests showed nearly a steadily linear increase for latency levels, as the quantity of senders and receivers rose. While these results, on themselves, do not seem surprising, they do showcase a different effect to those gathered using external microservices, at least in regards to the broadcaster mode and requests being utilized.

5.3 Discussion and evaluation

The results produced by the set of test cases were able to confirm some predetermined assumptions as well as reveal new insights into the prototype. The following discussion of the test results has been divided into areas that were deemed most significant based on the findings. These were the messages themselves, the two queue modes, the senders and receivers, as well as the requests and responses. There is also an additional subsection

dedicated to a phenomena of unique latency spikes, that was observed during some test runs.

5.3.1 Messages

The first observation, based on test 1 results, was the noticeable impact of message size as a factor on average latency. While not having a particularly large impact, there seemed to be a relatively linear rise in latencies, when the message sizes were increased, at least on larger microservice amounts. The exception here was the set of test runs, that were executed with 4 kB sized messages. For an unknown reason, these specific test runs regularly exceeded the average latencies that were achieved with larger message sizes. The impact also seemed to increase as more microservices were included. To avoid these kind of issues, the message size was locked down at 50 B for the following tests. Test 2, on the other hand, showed that sending additional messages did not affect the results in a significant way. As such, this setting was deemed not as relevant to take into account when performing subsequent tests.

with test 3, the real limits of the prototype were starting to become highlighted. As previously presented, the increases in traffic rate, combined with a large number of microservices, caused the prototype fall behind in performance. There was a clear jump in overall latencies, when the sending delay of the prototype was lowered to the 0.1 - 0.2 seconds range and the microservice amounts exceeded 50. After this point, the prototype simply could not keep up with the raw amount of messages being passes through, forming a clear cutoff point for the prototype’s usefulness. This distinct of a limit, however, was only noticeable when using the broadcaster mode.

In the last two data sets of test 3, where the load balancer mode was utilized, no reoccurring changes could be noticed in average latency, not when the quantity of senders and receivers was increased, nor when the delay between sent messages was lowered. Any differences between test runs, seemed to occur randomly from test run to test run, on a relatively minor scale. All of the measured latencies were also consistently lower than those measured with test runs involving broadcaster mode.

At this point, it was clear that reducing the delay between sending messages would prove to be an issue in certain instances, at least when broadcaster mode was utilized. Also, it had become obvious, that the two queue modes were significantly different, not just in function but also in performance.

5.3.2 Queue modes

It was clear from the start, that the broadcaster mode would be more taxing than the load balancer mode in every test case. Since broadcaster mode requires each message to be sent everywhere, this naturally requires more operations that need to be executed, which equals to additional stress for the prototype. When the implemented microservices are placed in different VMs, they are able to divide the stress and achieve better results, as the traffic increased. Due to this, the external services became much more beneficial on average when the broadcaster mode was being utilized. Even the natural benefits of a lesser transfer distance became insignificant after a certain point. The core of the issue here seems to be the specifications and processing capabilities of each machine. If the used machine would be powerful and flexible enough, each of these microservices could theoretically be run internally, even with the broadcaster mode. However, the less powerful the individual machines are, the more favorable distribution becomes.

While the main aim of test 4 was to see whether increasing the amounts of senders and receivers would produce different results, it also continued to showcase the significant advantage of the load balancer mode over the broadcaster mode, in terms of average latency. This, however, does not mean that broadcaster mode would be any less worthy of utilization. After all, both of these queue modes can be used to produce an almost opposite outcome, as far as traffic goes. Since broadcaster mode sends every message to every receiver, and load balancer mode sends each message only once, it naturally follows that the results are also significantly different. In a sense, when using broadcaster mode, the prototype is being run in a sort of worst-case scenario, whereas load balancer mode emphasizes significantly more favorable circumstances.

5.3.3 Senders and receivers

As far as the stress caused by the different microservices goes, the results came out as expected, in most cases. It seemed evident, that the quantity of active receivers would be significantly more impactful on latency, when broadcaster mode was used, since this meant that every message had to be sent to each of those receivers. With the results gathered from test 4, this was proven to be true. The collected results showed a nearly linear increase in latency, as the amount of receivers was increased. In comparison, the amount of senders seemed to have a comparably small impact, with the overall latency remaining on a similar level throughout the different test runs. As an additional observation, it seemed that increasing the quantity of senders at the same time as the receivers, led only to a small additional increase in latency, while retaining the linear nature of the tests, where only receivers were added. What was even more surprising, however, were the results gathered from the load balancer mode tests runs during test 4.

With the load balancer mode being utilized, it was so far noticed, that latencies seemed to stay relatively steady across the board, even when other setting were changed. This was until the number of receivers alone was increased, with the sender amount remaining at minimum. After the amount of 80 receivers was reached, the latencies started to suddenly shoot up, while remaining relatively steady prior to this point. After a more careful study of the test data, the reason for these unexpected results seems to be a small set of significant latency spikes, occurring when some of the messages were sent. It is, however, surprising that spikes as significant as these only seemed to occur when the number of senders was kept very small compared to the amount of receivers. Similar latency spikes were noticed in other tests as well, on a lesser scale, and the topic is discussed further at the end of this chapter.

As mentioned before, the location of the senders and receivers seemed to have a significant effect on the performance of the prototype, regardless of the queue mode used. When testing was first started, the assumption was, that when these microservices are running in the same machine as the N2NQ, it would lead to smaller latencies for each message, simply due to the shorter distances. While this could already be observed with test 3, the idea was further studied with test 5. Based on the tests, the assumption was seemingly

proven true, at least in the cases where load balancer mode was used, since it inherently caused less traffic. However, the results were not as simple with broadcaster mode, where the benefits of lesser distances were quickly overcome by the inability to compensate for the additional stress, when running the different microservice processes in the same VM as the N2NQ instance.

5.3.4 Requests and responses

When running test 6, the issues and limitations with requests and responses became quickly evident. With both of these, on their own, more than doubling average latency on broadcaster mode, as well as lesser increases on load balancer mode, the use of requests and responses should be limited as much as possible. The situation became even more unbearable with both of them on, resulting in an unsustainable operating environment already after 20 or 30 senders and receivers, with broadcaster mode and load balancer mode respectively. While similar issues were previously noticed when pushing internal microservices to the limit, this issue introduces itself early on, even when using external ones. Based on these results, it is safe to say that requests and responses should only be used in specific cases, with their benefits and drawbacks thoroughly evaluated, and only one at a time.

After performing test 7, it was also confirmed, that the far superior method latency wise, for achieving each message to get sent only once, was the use of the load balancer mode over the broadcaster mode with requests. This was not on its own unexpected, but the results proved the load balancer mode to being less taxing by quite a large margin. Still, it is worth repeating that both of these methods work in very different ways and whereas the load balancer mode sends out the messages randomly, requests can be used to organize them in a more controlled manner. Requests also provide additional features for message distribution and this test only measured their use from a specific perspective.

5.3.5 Latency spikes

During the early testing, it was observed that during some of the runs, significant spikes in latencies with individual messages could be measured. As mentioned before, these were

deemed so significant that average latencies were changed to utilize only the 95th percentile values. The specific reasons for each spike likely depends on a variety of reasons, but their impact seemed to increase with overall system stress.

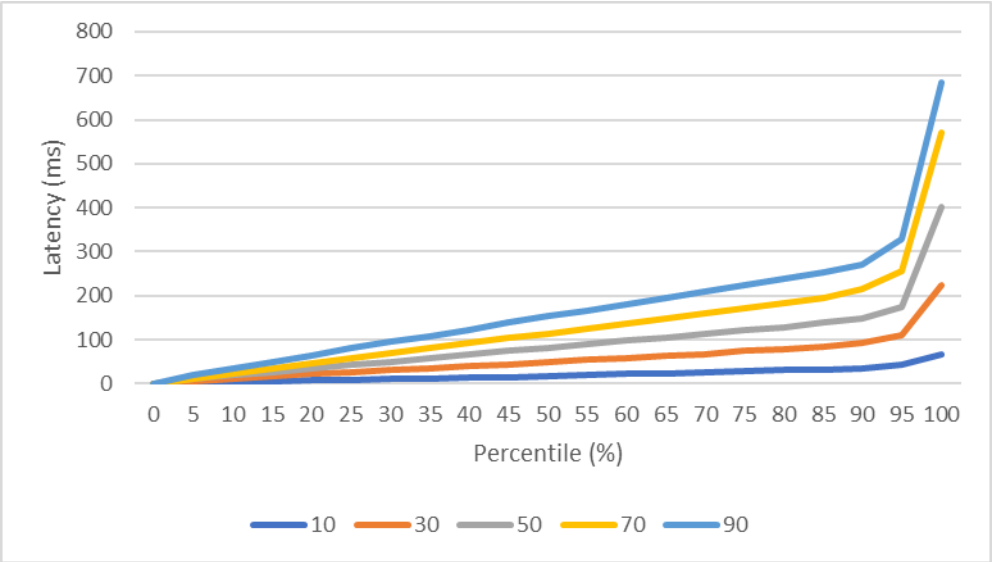

Figure 29. Latency by percentile, from test 5 with internal microservices.

Above in Figure 29, a set of latency percentiles from test 5 are presented. In the case of this figure, the broadcaster mode was utilized with all senders and receivers running in the same machine as the N2NQ. From the figure, it can be clearly seen, that the severity of the spikes increases as additional senders and receivers were added. Prior to these spikes, the latency seems to have increased in a mostly linear fashion throughout the different percentiles. There is, however, already a small noticeable change after the 90th percentile mark. At this point, the latencies are already starting break out of the steady linear increase. While this does not constitute a similar spike that appears after the 95th percentile, it is clearly visible. Interestingly, this kind of behavior cannot be notices when the implemented microservices are changed for external ones.

As Figure 30 below shows, unlike when using internal senders and receivers, the external microservices show no noticeable extraordinary increases in latency, before the 95th percentile. Prior to this point the increase in latency appears nearly linear. The actual spikes from 95th percentile onwards, however, do not seem to be in any significant way larger between the test runs with internal and external microservices. In fact, the difference

in average spikes between these two examples was measured to being less than 8 milliseconds, whereas the highest spikes themselves exceeded over 350 milliseconds. When observing latencies throughout the runtime, it became clear that most of these significant spikes were observable during the start of each test. This, however, did not, in itself, explain their origin.

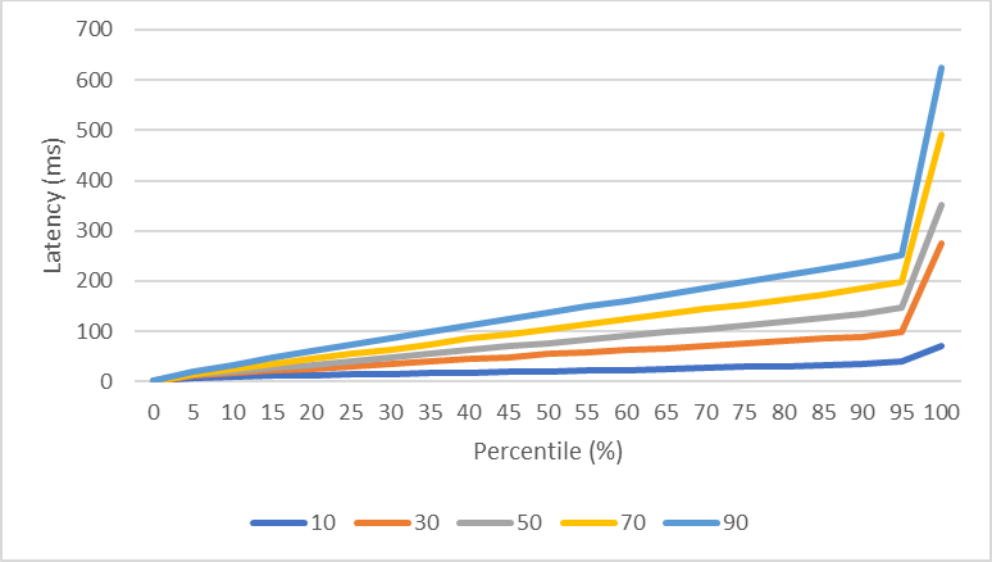

Figure 30. Latency by percentile, from test 5 with external microservices.

In addition to general increases in system stress, there could be other factors affecting these spikes as well. During the testing, it was noticed that in some instances the measurements showed noticeable variations from the norm, with additional spikes and overall higher latencies. Since the prototype has been implemented on a set of VMs running in a previously established cloud, it was assumed that this behavior could be the result of the cloud itself being under higher than usual stress. Since the cloud used was not designed for the specific use of this prototype, and was serving other VMs as well, unrelated to this thesis, it seemed possible that external factors could also be taxing the capabilities for operating this prototype. The problem comes with measuring these external factors, since they are involved with unrelated projects. Due to this, finding concrete evidence and causal links was not possible at the time of writing.

6 FUTURE WORK

During the development process, numerous ideas were raised about the testing possibilities regarding the N2NQ concept. Due to time and scope restrictions, only a part of these were ultimately included in the thesis and implemented into the prototype. The crux of this chapter is to present a set of possible ideas, that have yet to be utilized or tested, that could give directions where the testing of N2NQ could be taken in the future and how the current work could be expanded.

6.1 TCP vs. UDP

While TCP suited the requirements of this thesis well, it also proved to have its deficiencies. UDP is well-known to be a faster protocol in most use cases and the possibility for gaining additional latency improvements is intriguing. Implementing UDP in the current prototype should be relatively straightforward, with certain limitations. If the goal is to simply get a set of senders to transmit messages to a set of receivers, using the N2NQ as a middleware, UDP should work fine. However, if additional features, like requests and responses, need be kept operational, implementing UDP could become more difficult.

One of the reasons why UDP was not chosen for this thesis was its one-directional nature. With TCP, a steady connection must be made and upheld throughout the communication process, which allows for messages to be sent to both ways without additional difficulties [22]. This two-directional connection is utilized by the prototype, in the execution of requests and responses. With UDP, messages are being sent without any such continuous connections, meaning that for each implementation, messages can only be sent to a specific direction. This means that UDP cannot be directly used to replace TCP, without additional work in implementation.

For requests to work with UDP, separate sending and listening functionalities would have to be implemented, for both receivers and the N2NQ instance. Since UDP aims to remain completely oblivious about what happens to the messages once they are sent, these

listening and sending functionalities should be able to fully operate independently from each other. This could make running the prototype with requests significantly faster, since in the current TCP implementation the single connection is forced to wait and listen between every message sent. However, these benefits might not apply for every situation.

As has been demonstrated with the tests of the current prototype, increasing the number of receivers significantly reduces performance. While implementing requests with UDP, each receiver would require a separate listening and sending functionality, which would also likely increase their overall processing requirements compared to the current TCP version. As long as there are enough resources to go around and not too many simultaneous receivers, separating listening and sending functionality would likely still provide benefits over TCP. However, it would be interesting to see whether these additional requirements would, at some point, overcome the benefits as more microservices are added.

An even bigger issue would appear with implementing responses. Since the whole point of responses is to confirm that messages are being transmitted properly, this inherently conflicts with UDP's core ideals of retaining decoupling and autonomy. While this kind of two-way coupled discussion could still be implemented, there are questions whether it would serve any benefits over a TCP implementation, which naturally supports this kind of functionality.

6.2 Containers

Containers are an operating-system-level virtualization method, that allow for deployment and operating of distributed services. Containers allow for users to create and execute pieces of software, isolated from the surrounding environment, without having to deploy a separate virtual copy of an OS instance for each of them. Unlike VMs, each container shares the hosts OS kernel and thus do not require the entire full physical hardware to be virtualized, making them significantly lighter. In practice, containers are usually seen as an alternative for VMs, although both can also be made to coexist together. [6]

There are many advantages for using containers instead of VMs. Containers have significantly lower requirements for system resources, such as RAM and CPU cycles.

Containers also tend to take up less space, with tens of MBs compared to VMs' tens of GBs. In most cases, a single server could manage multiple times the number of similar applications on containers than on VMs. There is also evidence, that applications running on containers start noticeably faster, decreasing load times from minutes to seconds. [6]

In terms of microservices, containers have already been proven as a suitable alternative. The ability to build, deploy and start services faster is a significant benefit when operating an application with a microservice architecture. Independent containers are also easy to scale for their specific requirements and the overall scale of the application can be easily changed by modifying the amount of deployed container instances. Updating of applications can also be made simpler, as changing the specific subset of updatable containers can be done without disturbing the others. Containers also provide multiple different approaches for microservice deployment, since in addition to the traditional master-slave solutions, containers can also be created inside other containers, forming a nested architecture. [6, 7]

In the case of N2NQ, it would be interesting to see what kind of benefits could be gained from switching to the use of containers. The current prototype utilizes only three VMs, with both senders and receivers grouped together into larger processes. This simulates only a small subset of microservice systems and could be split into additional parts in the future. Instead of creating a large amount of new VMs, the prototype could be made to run on a set of containers, with possibly one for each individual microservice process. This could provide a more accurate testing environment, require significantly less system resources and space, and might even lead to some additional speed benefits.

6.3 Multi-cloud communication

As mentioned at the start of this thesis, the N2NQ concept has been envisioned as potentially suitable for a multi-cloud environment from the beginning, when it was first conceived. While this thesis limited the scope strictly to a single cloud, there are no inherent reasons why a more comprehensive implementation could not work, at least in theory. There are, however, multiple changes that would have to be made, so that a jump to multi-cloud communication could be achieved.

Let’s assume that the implementation of this expanded version would be based on the original multi-cloud concept figure, that was presented as Figure 4 in chapter 3. By viewing this figure, a few obvious differences, compared to the current implementation, instantly present themselves. First of all, the current prototype does not provide functionality for multiple N2NQ instances to communicate with each other, even within the same cloud. To make this possible, specifically designed interfaces would need to be implemented between the different N2NQ instances. In the complex set-up of Figure 4, a large amount of local N2NQ instances are presented. These should be able to gather and forward messages to local microservices, but also have the ability to communicate with each other. As previously mentioned, this functionality could be achieved with the use of a central N2NQ, that acts as a middleware between all the other N2NQ’s in the same cloud.

As showed in Figure 4, the idea of the central N2NQs involves them connecting to each of the local N2NQ instance within their respective cloud. None of the local N2NQs connect directly to each other and all of the messages, that are required outside of a local context, go through the central N2NQ. Once all of the messages have been centralized to a specific place, it is much easier to implement a connection to another cloud. As with the local N2NQs, the central instances would also use a set of four interfaces, two for the N2NQs found in the same cloud and two for the other central N2NQs found in other clouds. When a connection is established between two clouds, the main goal of the central N2NQs would be to share necessary messages to the subsequent clouds, as fast as possible. Each of these received messages would then be shared onwards to the local N2NQs which would forward them to their local microservices.

To make this kind of set-up work, a lot more processing power and faster connections might be required, as transmitting between clouds will likely increase latency due to increased distance. Each central N2NQ would probably also experience stress levels that are many times higher than those found in the local instances, since all of the messages from the local N2NQs might get sent to them, at least in a worst-case scenario. Due to this, a significant boost in available resources should be provided. If the central N2NQ cannot keep up with the stress, the performance of the entire system could be threatened. Achieving low-latencies also becomes significantly more difficult when operating on a

larger scale, especially with multiple clouds. With each message having to bounce between multiple different N2NQs, finding the fastest transfer methods and utilization of powerful servers becomes essential, or else low-latencies might not be achieved.

7 SUMMARY

Based on the original N2NQ concept, an operable prototype was developed to test its practical capabilities. The prototype was designed to work on a set of VMs operating within a shared cloud environment. A separate GUI was also developed, to control the operating of the prototype and manage the settings it was run with. The testing procedure itself was divided into seven unique test cases, each designed to focus on specific areas of the prototype and the features it offered, with a specific focus on studying what kind of latencies could be achieved and which factors affected them.

The executed tests provided a diverse set of results on the nature of latency achievable in the prototype. Between test runs, the average latencies for messages ranged from hundreds of microseconds to hundreds of milliseconds. While the message amount did not impact latency, their transfer rate did, especially when the sending and receiving microservices were running in the same VM as the N2NQ instance. The most significant impact was caused by overloading the system with excessive traffic, when internal microservices were being utilized, as well as the simultaneous use of requests and responses. In both of these cases, the N2NQ instance failed to keep up with the load of traffic, resulting in constantly rising latencies. Other latency wise impactful settings were increasing the number of senders and receivers and the use of a specific queue mode, which defined whether messages were removed instantly after being sent or transferred to each of the receivers.

There are many ways how the current prototype could be refined, to facilitate the further testing of it, as well as N2NQ itself as a concept. The first option would be to create a UDP based communication mechanism as an alternative for the current TCP implementation. This could be beneficial, as UDP is generally regarded as a faster message transmission protocol. Another possibility would be the utilization of containers, instead of VMs, for deployment and running of microservices. In the case of this prototype, each sender and receiver could be possibly placed separately to individual containers, due to them requiring significantly less resources and memory than similar VMs. There is also the possibility of testing the concept as a more complex implementation, within a multi-could environment, with specific N2NQ instances communicating from cloud to cloud.

REFERENCES

1. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M., Microservice Architecture: Aligning Principles, Practices, and Culture, 1st edition, O'Reilly Media, Inc., USA, 2016.
2. Richards, M., Microservices vs. Service-Oriented Architecture, 1st edition, O'Reilly Media, Inc., USA, 2015.
3. Williams, D., Integration Patterns and Anti-Patterns for Microservices Architectures, Williams & Garcia, December 2015. From: https://www.slideshare.net/Apcera/integration-patterns-and-antipatterns-for-microservices-architectures.
4. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., Microservices: yesterday, today, and tomorrow, Present and Ulterior Software Engineering, Springer, 2017, pp. 195-216.
5. Confluent, Inc., Microservices in the Apache Kafka Ecosystem, 2017. From: https://www.confluent.io/resources/microservices-in-the-apache-kafka-ecosystem.
6. Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., Steinder, M., Performance Evaluation of Microservices Architectures using Containers, *IEEE 14th International Symposium on Network Computing and Applications (NCA)*, September 28-30, Cambridge, MA, USA, 2015, pp. 27-34.
7. Venugopal, M.V.L.N., Containerized Microservices architecture, *International Journal of Engineering and Computer Science (IJECS)*, Vol. 6, No. 11, November 2017, pp. 23199-23208.
8. Hohpe, G., Woolf, B., Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, 1st edition, Addison-Wesley Professional, USA, 2003.
9. Johansson, L., What is message queueing?, CloudAMQP, December 2014. From: https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html.
10. Hunkeler, U., Truong, H.L., Stanford-Clark, A., MQTT-S — A Publish/Subscribe Protocol For Wireless Sensor Networks, *3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE)*, January 6-10, Bangalore, India, 2008, pp. 791-798.

11. Eugster, P.Th., Felber, P.A., Guerraoui, R., Kermarrec, A.-M., The Many Faces of Publish/Subscribe, *ACM Computing Surveys (CSUR)*, Vol. 35, No. 2, June 2003, pp. 114-131.
12. Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J., Stein, J., Building a Replicated Logging System with Apache Kafka, *Proceedings of the Very Large Data Bases Endowment (VLDB)*, Vol. 8, No. 12, August 2015, pp. 1654-1655.
13. Nguyen, C.N., Kim, J.-S., Hwang, S., KOHA: Building a Kafka-Based Distributed Queue System on the Fly in a Hadoop cluster, *IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*)*, September 12-16, Augsburg, Germany, 2016, pp. 48-53.
14. HiveMQ, MQTT Essentials: Part 1 – Introducing MQTT, Blog, 2015. From: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt.
15. Luzuriaga, J.E., Perez, M., Boronat, P., Cano, J.C., Calafate, C., Manzoni, P., Improving MQTT Data Delivery in Mobile Scenarios: Results from a Realistic Testbed, *Mobile Information Systems (MOB INF SYST)*, Vol. 2016, No. 1, July 2016.
16. Pardo-Castellote, G., OMG Data Distribution Service: Real-Time Publish/Subscribe Becomes a Standard, Industry Insight, January 2005. From: https://www.rti.com/hubfs/docs/reprint_rti.pdf.
17. Pardo-Castellote, G., OMG Data-Distribution Service: Architectural Overview, *23rd International Conference on Distributed Computing Systems Workshops*, May 19-22, Rhode Island, USA, 2003, pp. 200-206.
18. Schneider, S., What's The Difference Between DDS And AMQP?, Electronic Design, April 2013. From: http://www.electronicdesign.com/embedded/what-s-difference-between-dds-and-amqp.
19. Kerner, S.M., Intel's Data Plane Development Kit Now a Linux Foundation Project, Enterprise Networking Planet, April 2017. From: http://www.enterprisenetworkingplanet.com/netsp/intels-data-plane-development-kit-now-a-linux-foundation-project.html.
20. Shrut, A., DPDK for Layman, September 2015. From: https://www.linkedin.com/pulse/dpdk-layman-aayush-shrut.

21. Scholz, D., A look at Intel's Dataplane Development Kit, *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, August 1, Munich, Germany, 2014, pp. 115-122.
22. Kumar, S., Rai, S., Survey on Transport Layer Protocols: TCP & UDP, *International Journal of Computer Applications (IJCA)*, Vol. 46, No. 7, May 2012, pp. 20-25.
23. Rad, M.R.N, Kourdy, R., TCP and UDP Performance comparison In Network on Chip, *Journal of Computing*, Vol. 4, No. 4, April 2012, pp. 14-18.
24. Andrews, J.G., Buzzi, S., Choi, W., Hanly, S.V., Lozano, A., Soong, A.C.K., Zhang, J.C., What Will 5G Be?, *IEEE Journal on Selected Areas in Communications (J-SAC)*, Vol. 32, No. 6, June 2014, pp. 1065-1082.
25. Lee, H., Concept and Characteristics of 5G Mobile Communication Systems, Netmanias.com, Blog, January 2015. From: https://www.netmanias.com/en/post/blog/7109/5g-iot/concept-and-characteristics-of-5g-mobile-communication-systems-1.
26. 5G Use Cases and Requirements White Paper, Nokia, 2016.
27. Gifrin, C., Simplify Your Pub/Sub Messaging with Amazon SNS Message Filtering, AWS Compute Blog, November 2017. From: https://aws.amazon.com/blogs/compute/simplify-pubsub-messaging-with-amazon-sns-message-filtering/.
28. Qt wiki, About Qt, February 2017. From: https://wiki.qt.io/About_Qt.
29. Mills, D.L., Internet Time Synchronization: The Network Time Protocol, *IEEE Transactions on Communications*, Vol. 39, No. 10, October 1991, pp. 1482-1493.