Analyzing parameter sets for RabbitMQ and Apache Kafka on a cloud platform

Amir Rabiee
Master of Science - Software Engineering of Distributed Systems

19-6-2018

Supervisor - Lars Kroll
Examiner - Prof. Seif Haridi

Abstract

Applications found in both large enterprises and small needs a communication method in order to meet criterias of scalability and durability. There are many communication methods out there but one of the most well used are the usage of message queues. There exists a plethora of many different types of message queues all unique regarding their design and implementation, this thesis tests two of them, Apache Kafka and RabbitMQ.

The experiments conducted are focused on two primary metrics, latency and throughput with secondary metrics such as disk usage and CPU usage. The parameters chosen for both RabbitMQ and Kafka is optimized with focus on the primary metrics. Moreover the experiments conducted are tested on a cloud platform, Amazon Web Services because they ran on a platform as a service called CloudKafka and CloudAMQP.

The results show that Kafka outshines RabbitMQ when it comes to throughput and latency but it also shows the impact that both Kafka and RabbitMQ has when it comes to the amount of written data, with RabbitMQ being the most efficient in terms of quantity of data being written while on the other hand being more CPU-heavy than Kafka.

Keywords: Kafka, RabbitMQ, throughput, latency, cloud platform, testing

Sammanfattning

Applikationer som finns i både komplexa och icke-komplexa system behöver en kommunikationsmetod för att möta kraven när det kommer till skalbarhet och uthållighet. Det finns väldigt många olika typer av kommunikationsmetoder men en av de mest välanvända är meddelandeköer. Det finns ett väldigt stort utbud av meddelandeköer som alla är unika med deras egna design och implementation, den här rapporten testar två av dem, Apache Kafka och RabbitMQ.

Experimenten som utfördes fokuserades på två huvudmätetal, latens och genomflöde med sekundära mätetal som minnesanvändning och CPU-användning. Parametrarna utvalda för både RabbitMQ och Kafka försöker optimera dessa mätetal. Experimeten utfördes på en molnplattform, Amazon Web Services eftersom tjänsterna existerar på den plattformen.

Resultaten visar att Kafka är tydligt bättre än RabbitMQ när det kommer till genomflödet och latensen men påvisar också effekten av både RabbitMQ och Kafka har på hur mycket data som skrivits där RabbitMQ är mer optimerat än Kafka men påvisar också att RabbitMQ är mer CPU-belastad.

Nyckelord: Apache Kafka, RabbitMQ, genomflöde, latens, molnplattform, testning

Contents

Chapter 1

Introduction

The world of applications and the data being generated and transfered to and from them are constantly evolving in a fast paced manner. The amount of data generated over the Internet and the applications running on it are estimated for 2020 to hit over 40 trillion gigabytes [1].

The applications handling this data has requirements that needs to be met such as having high reliability and high availability. Because of the high demands for such requirements a natural outcome of this is to build a distributed system that can support these applications whether they be a load balancing system or a game application or anything in between. [2, p. 1].

The evolving of distributed systems stems from the relative cheap hardware commodity that one has been able to utilize to build networks of computers communicating together for a specific purpose. These systems are in comparison to the *client-server* architecture constructed of different applications running on multiple separated machines. Because of the inherit nature of having multiple machines coordinating together for a common task, an innately advantage for them is that distributed systems are more scalable, reliable and faster when architectured correctly in comparison to a *client-server* model. The advantages with these systems comes with a cost, as designing, building and debugging distributed systems are more complicated than systems running on only one machine [2, p. 2].

In order to assess the scalable part of a distributed system, appropriate measures have to be taken, to easily meet the need when the communication between different machines and their applications are put to the test. To help facilitate the problems that can occur when an application on one machine tries to communicate with a different application on another machine one can use **message queues** and **message brokers** [3, p. 2].

The message brokers works in symbiosis with the message queues to help deliver messages sent from different destinations and route them according to the correct route [4, p. 326].

1.1 Background

The distributed systems that are developed to meet the requirements of having high availability and reliability are built upon some abstract message form being sent from one process located on one machine to another process on a different machine. Therefore an inherit attribute of a distributed system is that it needs an architecture or system that can distribute messages in order to achieve higher scalability and reliability.

1.1.1 Message oriented middleware

The architecture used, that strives to fulfil the requirements, is called *message-oriented middleware* (MOM), this middleware is built on the basis of an asynchronous interaction model which enables users to not be blocked after sending a message, instead they continue processing with their execution [5, p. 4]. More importantly a message oriented middleware is used as a basis for distributed communication between processes without having to adapt the source system to the destination system. This architecture is illustrated in Figure 1.1 where the apps in this context refers to the various systems using the MOM.

A central concept and structure when using a message oriented middleware is the usage of **message queues**, these queues are used to store messages on the MOM platform. The systems using the MOM platform will in turn use the queues to send and receive messages through them. These queues have many configurable attributes, which include the name, size, the sorting algorithm for the queue and so forth [5, p. 7].

In order to efficiently distribute the messages to different queues one has to resort to a **message broker**, these message brokers can be seen as an architectural pattern according to [6, p. 288], the broker is responsible for translating different message data formats between applications which abstracts the routing from one destination to another.

An important attribute of message brokers is their usage of message queue protocols, there are many different types of protocols that can

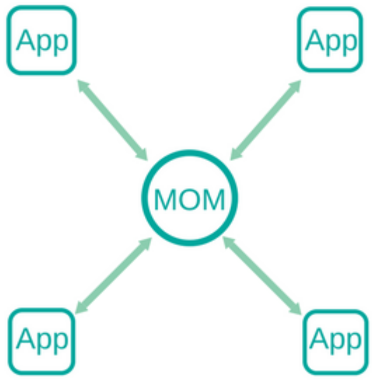

Figure 1.1: Structure of message oriented middleware.

be used such as **Streaming Text Oriented Messaging Protocol** (STOMP), **Extensible Messaging and Presence Protocol** (XMPP), **Message Queueing Telemetry Transport** (MQTT) and **OpenWire** and more [7, p.3].

The unique capabilities of the MOM-model comes from the messaging models that it uses, there are mainly two messaging models to be found, **point-to-point** and **publish/subscribe**.
The point-to-point model provides a communication link between the producer and consumer with the usage of a message queue, the consuming clients processes messages from the queue with the requirement of having only one receiver consuming the message albeit not being a strict requirement [5, p. 9], these messages are delivered **exactly once**.
In the publish/subscribe model, the producer produces a message to a specific topic, the consumers interested in these messages will subscribe to the topic, and thereafter be routed by a publish/subscribe engine.

An intuitive metaphor that can be used to understand the usage of the MOM-model is to see it as a post terminal where the message brokers are the persons delivering the posts to the right post code area which in this case are the message queues.

1.2 Problem

With a plethora of different message brokers available to help with implementing a message oriented middleware, choosing one is a multifaceted question that has many different aspects that need to be taken in consideration. Every message broker has their own design and implementation goals and can therefore be used for different purposes and situations. An overview of some message brokers and the protocols supported can be seen in Table 1.1.

Table 1.1: Message brokers and their supported protocols

Message broker	Protocols supported
Apache Kafka	Uses own protocol over TCP
RabbitMQ	AMQP, STOMP, MQTT,
ActiveMQ	AMQP,XMPP, MQTT, OpenWire

1.2.1 84codes, CloudKarafka, CloudAMQP

The message brokers found in Table 1.1 are widely used and can be deployed on different server architectures and platforms [8], the company **84codes** offers two different services called **CloudKarafka** and **CloudAMQP** [9][10]. CloudAMQP is a RabbitMQ-focused implementation and CloudKarafka is an Apache Kafka solution, these two services are hosted on different cloud platforms such as Amazon Web Services, Google Cloud, Microsoft Azure etc.

These two platforms as a service needs to be tested on their enqueueing performance in order to get a deeper understanding of the underlying mechanisms behind the enqueueing decisions when sending a message from a producer to a consumer. With this in mind the main problem adressed by this thesis work is: Performance testing of Apache Kafka and RabbitMQ on Amazon Web Services to be able to answer the question and quantify which situations to use RabbitMQ and Kafka on a cloud platform?

1.3 Purpose

Because of the intricacy of the different message brokers and their corresponding protocols they use it is a relative difficulty in grasping both the fine grained differences between them as well as the coarse grained. This thesis discusses and shows an overview of the available messaging middleware solutions and aims to validate and verify the enqueueing performance for two message brokers. The enqueueing performance focuses on the throughput aspect versus the latency, moreover the thesis focuses on the resource usage of both the message brokers such as CPU and memory during load.

Furthermore the two different message brokers RabbitMQ and Apache Kafka is a debatable topic on deciding which one to use [11], this thesis work will try to shed a light on this subject, as well as focus on testing the two platform-as-a-service (PaaS) CloudKarafka and CloudAMQP on Amazon Web Services.

The thesis work will present the designed testing experiments and the results of them, in order to visualize the performance of the two message brokers running on the cloud platform Amazon Web Service.

1.4 Goal

The goals of this project is presented below:

- Design experiments for testing the enqueueing performance for Apache Kafka and RabbitMQ.
- Compare the results from each experiment and discuss the findings with regards to the the cloud platform and the respective message broker.
- Evaluate with a 95th percentile for sending and processing messages.
- Use the project as reference material when analyzing Apache Kafka and RabbitMQ for their enqueuing performance.

These goals presented lays the foundation for this thesis work and the results derived from the goals can be further used as a reference point for when to use RabbitMQ over Kafka and vice versa.

1.4.1 Benefits, Ethics and Sustainability

With the amount of data being generated in the present day which can be found in many enterprises one has to think of the ethical issues and aspects that can arise with processing and storing this much information and what the possibilities are with extracting valuable data from this content.

This thesis work is not focused on the data itself that is being stored, sent and processed, but rather the infrastructures which utilizes the communication passages of messages being sent from one producer to a consumer. Nonetheless the above mentioned aspects are important to discuss, because from these massive data-sets being generated one can mine and extract patterns and human behaviour [12], which in turn can be used to target more aggressive advertisements to different consumer groups.

Moreover another important aspect to be brought up is the privacy and security of peoples personal information being stored which can be exposed and used for malicious intent [13], this will be remedied to a degree with the introduction to the new changes in the General Data Protection Registration that comes into effect May 2018 [14] .

With the usage of large cloud platforms and their appropriate message brokers that is used to send millions of messages between one destination to another, one has to think of the incumbent storage solutions for the data, which in this case has resulted in the building of large datacenters. These datacenters consumes massive amount of electricity, up to an amount of several megawatts [15], in comparison to a toaster that uses about 1 kilowatt.

The company 84codes and their services holds a greater standpoint on the ethical aspect because both CloudKarafka and CloudAMQP is used by companies all over the world for sending data and therefore the ultimate responsibility of security lays on 84codes and their strategies and design decisions to keep the data intact and not exposed to third-parties or other non-authorized parties.

The main benefitters of this thesis work are those standing at a crossroads of choosing a message broker for their platform or parties interested in a quantitative analysis focused on the enqueuing performance of two message brokers, Apache Kafka and RabbitMQ on a cloud platform such as Amazon Web Services.

1.5 Methodology

The focus of this thesis work is to analyze and test the enqueuing performance of two different message brokers, and with that in mind, one has to think of the different methodologies and research methods that are to avail and that are the most suitable to conduct such a thesis work. The fundamental choosings of a research method is based on either a *quantitative* or *qualitative* method, both of these have different goals and can be roughly divided into either a numerical or non-numerical project [16, p. 3].

The quantitative research method focuses on having a problem or hypothesis that can be measured with statistics and validified as well as verified, a qualitative research on the other hand focuses on opinions and behaviours to reach a conclusion and to form a hypothesis.

For this project the most suitable research method is of quantitative form because of the experiments and tests to be run gathers numerical data.

Another important aspect to be chosen for the thesis work is the *philosophical assumption* that can be made and there are several school of thoughts to be considered. Firstly the *positivism* element relies on the independency between the observer and the system to be observed as well from the tools to be measured with. Another philosophical school is the *realistic*, which collects data from observed phenomenons and thereafter develop knowledge. Thirdly a *criticalism* element is the one which focuses on learning how users can affect different types of computer systems. [16, p. 4]

There are several others philosophical principles but for this project the most fitting ones was to use a combination of both the positivism as well as the realistic.

Moreover to continue with the experimental testing of the enqueuing performance of the different message brokers a research method that fits the requirements of the thesis work had to be chosen. There are mainly two divisions of research methods, a *experimental* or a *non-experimental* research method.

Because of the nature of the thesis residing in experimenting with the correlation of variable changes and their relationship between one another in order to see how the message brokers becomes affected an obvious choosing would be a experimental research methodology. An *analytical* research method based on previous testing of both RabbitMQ and Apache Kafka is also showcased for more a comprehensive conclusion. [16, p. 4]

1.6 Delimitations

This report will not go in to depth of how the operating system affects the message queues because the workings used for memory paging are to complex to track and needs a further explanation depending on what operating system being used. Furthermore this thesis will not explore how different protocols can affect the performance of the brokers.

1.7 Outline

In text, describe what is presented in Chapters 2 and forward. Exclude the first chapter and references as well as appendix.

- **Chapter 2** shows a more in-depth technical background of Apache Kafka and RabbitMQ and their protocols.
- **Chapter 3** presents the research methodologies of the thesis work with the appropriate statistical tools.
- **Chapter 4** will present the different experiments conducted
- **Chapter 5** will present and visualize the results from the experiments.
- **Chapter 6** will discuss the results and the conclusions that can be drawn from the thesis work.

Chapter 2

Background

In order for two different applications to communicate with each other a mutual interface must be agreed upon. With this interface a communication protocol and a message template must be chosen such that an exchange can happen between them. There are many different ways of this being conducted, one can define different types of schemas in a programming language or rather let two developers agree upon a request of some sort, containing some arbitrary identifaction. Provided this mutual agreement between the two applications then it is of no importance for these applications to know the intricacies of one anothers system. Furthermore the programming language and framework can over the years change for the application but as long as the mutual interface stays firm between these two applications they will always be able to communicate with each other which in turn results in lower coupling between systems.

To further attain a lower coupling one can introduce *messaging systems* or *message-oriented middleware* which are interchangeable. The messaging system enables the communication between sender and receiver to be less conformative in a way where it is not necessary for the sender to preserve information on how many instances of receivers there are, where they are located or whether they are active [17, p. 3]. The message system is responsible for the coordination of message passing between the sender and receiver and has therefore a primary purpose of safeguarding the communication between two parties the same way a database has the task of storing each data record safely and persistent [4, p. 14].

Because of the inherit unreliability of networks and computers the reason message systems exists is to help mitigate the problem of a message being lost after being sent when a receiver has gone down. The message system in this case will try to resubmit the message until it is successful.

There are different types of models that a message system can use but there are mainly three different types of communication models, a point-to-point, publish/subscribe or a hybrid of those two.

2.1 Point-to-Point

> *"Alexandra walks into the post office to send a parcel to Adam. She walks up to the counter and hands the teller the parcel. The teller places the parcel behind the counter and gives Alexandra a receipt. Adam does not need to be at home at the moment that the parcel is sent. Alexandra trusts that the parcel will be delivered to Adam at some point in the future, and is free to carry on with the rest of her day. At some point later, Adam receives the parcel."* [17, p. 3]

The above mentioned quote demonstrates the mechanism of *point-to-point* communication where it is implemented with queues that uses a first in first out schema. This leads to only one of the subscribed consumers receiving the message, this can be seen in Figure 2.1.

The usage of point-to-point communication is found in applications where it is only of importance that you execute something once for example a load balancer or transfering money from one account to another.

When it comes to queues one has to discuss the attributes of them, more specifically *persistence* vs *durability*. The persistence attribute focuses on if a failure happens during message processing that the message is not vanished next time it is processed. This is done by storing the message to some other form than in-memory for example a temporary folder, a database or a file etc [18].

The durability aspect is when a message is sent to a queue and the queue is offline, and there after the queue comes back online, it is of importance that the queue fetches the messages that was lost during the downtime.

With persistence the reliability increases but at the expense of performance and it all boils down to design choices for the message systems to choose how many of the messages should be persisted and in what ways.

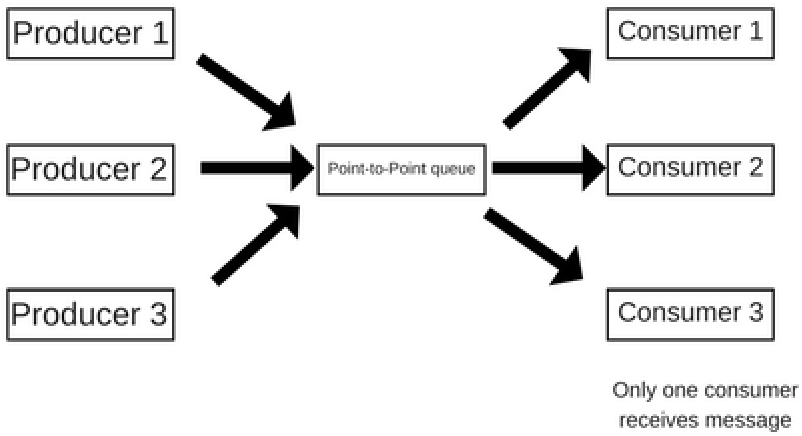

Figure 2.1: A point-to-point communication domain.

2.2 Publish/Subscribe

> *"Gabriella dials in to a conference call. While she is connected, she hears everything that the speaker is saying, along with the rest of the call participants. When she disconnects, she misses out on what is said. On reconnecting, she continues to hear what is being said."* [17, p. 4]

The quote above demonstrates the publish/subscribe interaction scheme. In this case Gabriella can connect herself up to a conference call and receive information from the speaker and the person speaking is not concerned with how many listeners there are. The system, in this case, the conference call, assures that anyone connected to the call will receive the information given. Gabriella in this scenario is the *subscriber* and the speaker the *publisher*.

This type of messaging architecture can be implemented with the help of *topics*, a topic can be seen as an event from which subscribers are interested in and whenever a message is sent to a topic from a publisher all the subscribers interested in such a topic becomes notified of the update. These events being sent to a topic are sent in a asynchronous manner and the strengths that lie within this type of architecture means that it can seperate the dimensionalities of *temporal*, *spatial* and *synchronization* from each other [19, p. 1]. This architecture is illustrated in Figure 2.2.

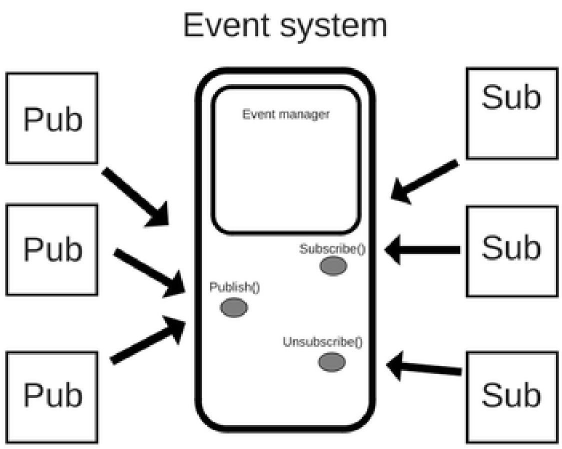

Figure 2.2: A publisher/subscriber domain.

The publishers can publish to their topics and the subscribers can either subscribe or unsubscribe from the topic. The three dimensionalities of the publish/subscriber domain helps to understand how a system with such a construct can enhance the messages passing from one producer to another consumer.

With the *spatial* dimensionality, the requirement of two interacting parties needing to know each other is not present. Because the publishers needs only to publish their data/message to a topic with an event service as a middle hand and the subscribers need only to interact with event service [19, p. 2]. The publishers will also not keep any data connected to the subscribers and does not know how many of the subscribers there are, in the same way that the subscribers does not know how many of the publishers exist.

The *temporal* dimensionality is that there are no requirements of both the publisher and subscriber to be active at the same time, that is, the publisher can send events to a topic while a subscriber is offline which in turn infers that a subscriber can get an event that was sent after the publisher went offline.

The *synchronization* dimensionality means that a publisher does not need to wait for a subscriber to process the event in the topic in order to continue with the execution. This works in unity with how a subscriber can get notified asynchronously after doing some arbitrary concurrent work.

A secondary type of implementation is the *content-based* publish/subscribe model which can be seen as an extension of a topic based approach. What differs a content-based from a topic-based is that one is not bound to an already defined schema such as a topic name but rather to the attributes of the events themselves [20, p. 4]. To be able to filter out certain events from a topic one can use a subscription language based on constraints of logical operators such as or, and, not etc [19, p. 9].

2.3 Communication protocols

There are many types of communication protocols that can be used when you have to send a message from one destination to another. These protocols has their own design goals and use cases that are more fitting than others and deciding on one protocol to use is a challenge. There are several use cases to think of such as, how scalable is the implementation, how many users will be sending messages, how reliable is sending one message and what happens if the messages do not get delivered etc. These are some of the aspects for the developer to dwell on and as such leaving them with protocols such as AMQP, XMPP, STOMP and MQTT.

2.3.1 Advanced Message Queueing Protocol

Advanced Message Queuing Protocol (AMQP) was initiated at the bank of JPMorgan-Chase with the goal of developing a protocol that provided high durability during intense volume messaging with a high degree of interoperability. This was of high importance in the environment of banking because there is an economic impact if a message is delayed, lost or processed incorrectly [21].

AMQP provides a rich set of features for messaging with a topic-based publish/subscribe domain messaging, flexible routing, security etc and is used by large companies that process over billion of messages a day ranging from JPMorgan Chase, NASA and Google [22].

The intricacies of how the AMQP protocol model is designed can be seen in Figure 2.3.

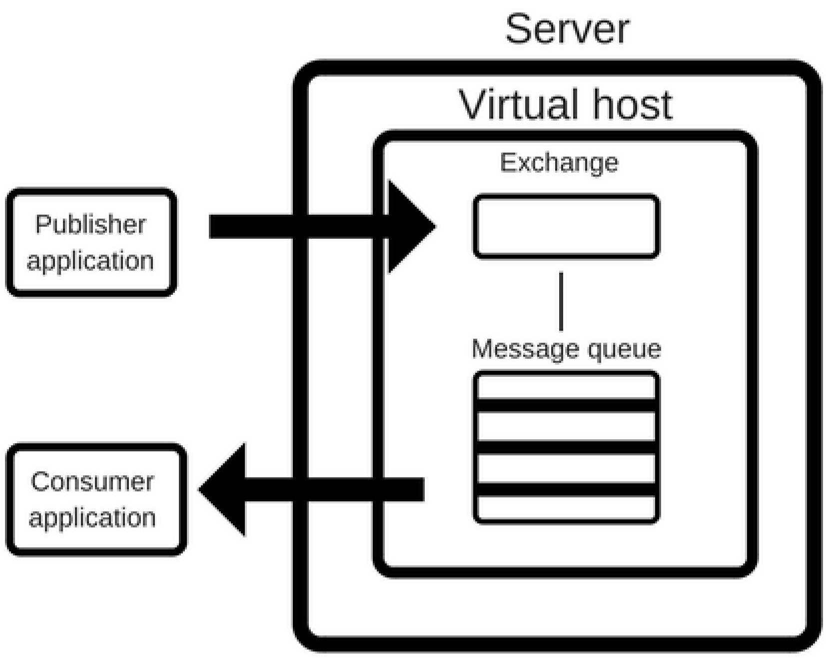

Figure 2.3: AMQP Protocol model

The exchange in Figure 2.3 accepts messages from producers and routes them to message queues, and the message queues stores the messages and forwards them to the consumer application [23].

The message queues in AMQP is defined as **weak FIFO** because if there exists multiple readers of a queue the one with the highest priority will take the message before the others. A message queue has the following attributes which can be configured,

- Name - Name of the queue.
- Durable - If the message queue can lose a message or not.
- Exclusive - that is if the message queue will be deleted after connection is closed.
- Auto-delete - the message queue can delete itself after the last consumer has unsubscribed.

In order to determine which queue to route a specific message from an exchange, a binding is used, this binding is determined with help of a routing key [24, p. 50].

There are several different types of exchanges found in AMQP, there is the direct type, the fan-out exchange type, topic and lastly the headers exchange type.

Direct type

This exchange type will bind a queue to the exchange using the routing key K, if a publisher sends a message to the Exchange with the routing key R, where K = R then the message is passed to the queue.

Fan-out

This exchange type will not bind any queue to an argument and therefore all the messages sent from a publisher will be sent to every queue.

Topic

This exchange will bind a queue to an exchange using a routing pattern P, a publisher will send a message with a routing key R, if R = P then the message is passed to the queue. The match will be determined for routing keys R that contain zero or more words, where each word is delimited with a dot. The routing pattern P works in the same way as a regular expression pattern. This can be seen in figure 2.4.

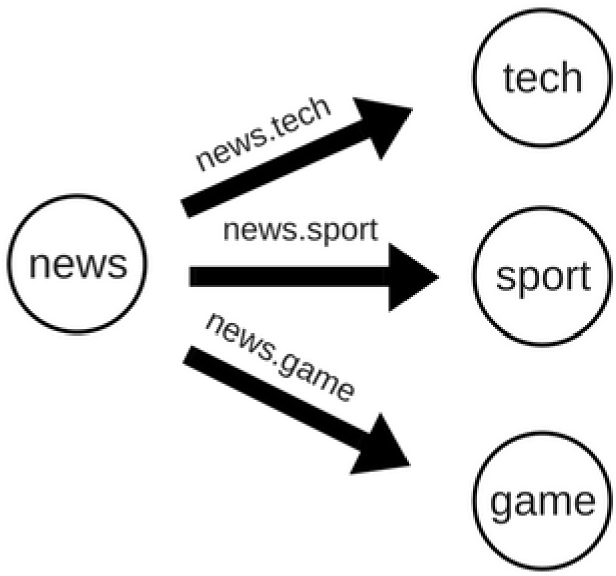

Figure 2.4: Topic exchange type

Headers

The headers exchange type will prioritize how the header of the message looks like and in turn ignores the routing key.

2.3.2 Extensible Messaging and Presence Protocol

The Extensible Messaging and Presence Protocol (XMPP) technologies were invented because of the abundance of different client applications for instant messaging services. The XMPP is an open technology in the same way as the Hypertext Transfer Protocol (HTTP), the specifications of the XMPP puts focus on the protocols and data entities that are used for real-time asynchronous communication such as instant messaging and streaming. [25, p. 7]

XMPP makes use of the Extensible Markup Language (XML) to enable an exchangement of data from one point to another. The technologies of XMPP is based on a decentralized server-client architecture much alike how the world wide web is deployed, this means, if a message is sent from one destination, the initial message is sent to an XMPP server and thereafter sent to the receivers XMPP server and finally to the receiver.

To be able to send a message to a person located somewhere else, XMPP sets up a XML-stream to a server and there after the two clients can exchange mesages with the help of three so called "stanzas", <message/>, <presence/> and <iq/> which are XML-elements. These stanzas can be seen as a data packet and are routed differently depending on what stanza it is.

The <message/> is what is used for pushing data from one destination to another, and are used for instant messaging, alerts and notifications [25, p. 18]. The <presence/> is one of the key concepts of real-time communications because this stanza enables others to know if a certain domain is online and ready to be communicated with. The only way for a person to see that someone is online is with the help of a presence subscription which employs a publisch-subscribe method. The info/query stanza <iq/> is used for implementing a structure for two clients to send and receive requests in the same way GET, POST and PUT methods are used within the HTTP.

What differs the iq stanza from the message stanza is that the iq has only one payload that the receiver must reply with and is often used to process a request. For error handling the XMPP does not acknowledge all of the packets sent over a communication link and XMPP assumes that a message or stanza is always delivered unless an error is received [25, p. 24].

The difference between a stanza error and a regular message error is that a stanza error can be recovered while other messages results in the closing of the XML stream that was opened in the start.

2.3.3 Simple/Streaming Text Oriented Messaging Protocol

The Simple/Streaming Text Oriented Messaging Protocol (STOMP) is a simple message exchange protocol aimed for asynchronous messaging between entities with servers acting as a middlehand. This protocol is not a fully pledged protocol in the same way as other protocols such as AMQP or XMPP, instead STOMP adheres to a subset of the most common used message operations [26].

STOMP is loosely modeled on HTTP and is built on frames, these frames is made of three different components, primarily a command, a set of optional headers and body. A server that makes use of STOMP can be configured in many different ways because STOMP leaves the handling of message syntax to the servers and not in the protocol itself. This means that one can have different delivery rules for servers as well as for destination specific messages.

STOMP employs a publisher/subscriber model where the client can both be a producer by sending frame containing SEND as well as being a consumer, this is done by sending a SUBSCRIBE frame.

For error handling and to stop malicious actions such as exploiting memory weaknesses on the server STOMP allows the servers to put a threshhold on how many headers there are in a frame, the lengths of a header and size of the body in the frame. If any of these are exceeded the server has to send an ERROR frame back to the client.

2.3.4 Message Queue Telemetry Transport

The Message Queue Telemtry Transport (MQTT) is a lightweight messaging protocol with design goals aimed to an easy implementation standard, having a high quality of service data delivery and being lightweight and bandwith efficient [27, p. 6]. The protocol uses a publish/subscribe model and is aimed primarily for machine to machine communication, more specifically embedded devices with sensor data.

The messages that are sent with a MQTT protocol are lightweight because it only consists of a header of 2 bytes and a payload of maximum 256 MB and a Quality of Service level (QoS) [28]. There are 3 types quality of service levels which are listed below

1. Level 0 - Employs a at-most-onec semantic where the publisher sends a message without an acknowledgement and where the broker does not save the message, more commonly used for sending non-critical messages.

2. Level 1 - Employs an at-least-once semantic where the publisher receives an acknowledgement atleast once from the intended recipient. This is done by sending a PUBACK message to the publishers

and until then the publisher will store the message and try to resend it. This type of message level could be used for shutting down nodes on different locations.

3. Level 2 - Employs an exactly-once semantic and is the most reliable level because it guarantees that the message is received, this is done by first sending a message stating that a level 2 message is inbound to the recipient, the recipient in this case replies that it is ready, the publisher relays the message and the recipient acknowledges it.

Moreover MQTT deploy something called a "Last Will and Testament" (LWT) for error handling, if a client disconnects abruptly which can be seen during power outages or unexepected network disturbances.

LWT is configured in the start for a client that connects to a broker, the broker will store it until a client disconnects abruptly, and broadcast the message to all subscribers that are connected to the topic that is published by the client. This ensures that the right precautions or actions are taken in the case of having a dead publisher.

Because of MQTT being a lightweight message protocol, the security aspect is flacking and the protocol does not include any security implementations of it its own as it is implemented on top of TCP, one is resorted to use SSL/TLS certifications on the client side for securing the traffic.

An indirect consequence of using SSL/TSL for encryptions is that it augments a significant overhead to the messages which in turn goes against the philosophy of MQTT being a lightweight protocol [29]. This is something that left for the developer to think about whether it is feasible to have more data being sent over the wire which can affect performance.

2.4 Apache Kafka

Apache Kafka is a distributed streaming platform used for processing streamed data, this central platform scales elastically instead of having an individual message broker for each application [30]. Kafka is also a system used for storaging as it replicates and persists data in infinite time, the data is stored in-order and is durable can be read deterministically [30, p. 4].

The messages written in Kafka are batch-processed, and not processed individually which is a design choice that favours throughput over latency, furthermore messages in Kafka are partitioned into *topics* and these are further classified into a set of *partitions*. The partitions contains messages that are augmented in an incremental way, and is read from lowest index to highest [30, p. 5].

The time-ordering of a message is only tied with one partition and not with rest of the partitions of a topic, each of these partitions can be allocated on different servers and is a key factor to enhancing scalability horizontally. This is illustrated in Figure 2.5

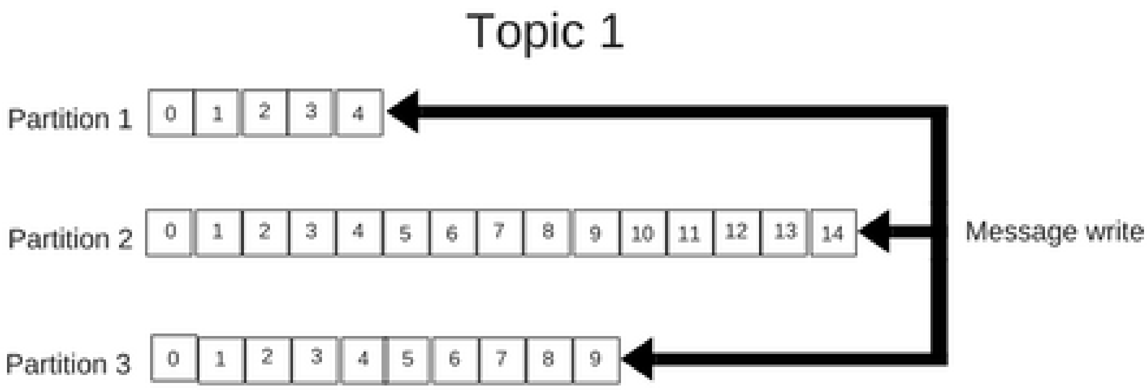

Figure 2.5: Topic partitioning

The messages within a partition is consumed by a reader and to keep track of which message has been read an *offset* is kept in the metadata in the case when a reader stops and starts reading later on.

Moreover a consumer cooperates with other consumers in a group to deplete a topic and there can only be one consumer from a consumer group for a partition. If a consumer fails the group will rebalance itself to take over the partition that was being consumed by the failed consumer [30, p. 7].

A Kafka server also called a *broker*, is responsible for commiting messages to the disk and giving offset measurements for producers, and responds to consumers requests for messages within a partition. These brokers will function with other brokers to form a *cluster*. This cluster deploys a leadership architecture where one broker will service as a leader for a partition and a so called *controller* is used for appointing leaders and to work as a failure detector. To increase the durability multiple brokers can be appointed to a partition and the partition is therefore replicated to other brokers.

2.4.1 Apache Zookeeper

To help maintain a cluster of Kafka servers, Apache Kafka utilizes Zookeeper to help keep track on metadata of the clusters and information regarding consumers. Zookeeper works as key-value manager that can help out with the synchronization aspects for Kafka such as leader election, crash detection, group membership management and metadata management [31, p. 11].

Zookeeper and Kafka works in symbiosis, where ZooKeeper tries to offer more control to issues that arises in a system with multiple clusters. Examples of failures that can happen when you have distributed coordination of multiple servers is that messages from one process to another process can be delayed, the clock synchronization of different servers can lead to incorrect decisions on when a certain message has arrived [31, p. 8].

2.5 RabbitMQ

RabbitMQ is a message broker that utilizes AMQP in an efficient and scalable way alongside other protocols. RabbitMQ is implemented in Erlang which uses the Actor-Model model. The Actor-Model is a conceptual model used for distributed computing and message passing, every entity in the model which are actors receives a message and acts upon them. These actors are seperated from one another and do not share memory, furthermore one actor can not change the state of another actor in a direct manner [8, p. 9][32]. The above mentioned reason is a key feature to why RabbitMQ is scalable and robust as all actors are considered independent.

RabbitMQ is in comparison to Apache Kafka mainly centered around and built upon AMQP which is presented in section 2.3.1. RabbitMQ makes use of the properties from AMQP and makes further extensions to the protocol.

The extension of the routing capabilities that RabbitMQ implements for AMQP is the ***Exchange-to-Exchange*** binding which means that you can bind one exchange to another in order to create a more complex and advanced message topology. Another binding is the ***Alternate-Exchange*** that works as a similarly to a wildcard matcher where there are no defined matching bindings or queues for certain types of messages. The last routing enhancement is the ***Sender-selected*** binding which mitigates the problem that AMQP has where it can not specify a specific receiver for a message.[33]

A fundamental difference of RabbitMQ to Apache Kafka is that RabbitMQ tries to keep all messages in-memory instead of persisting them to secondary memory such as a disk. In Apache Kafka the retention mechanism of keeping messages for a set of time is usually done by writing to a disk with regards to the partitons of a topic. In RabbitMQ consumers will consume messages directly and relies on a ***prefetch-limit*** which can be seen as a counter for how many messages that has been unread and is an indicator for a consumer that is starting to lag. This is a limitation to the scalement with RabbitMQ because this prefetch limiter will cut of the consumer if it hits the threshhold resulting in stacking of messages.

The deliver semantic of RabbitMQ for the message queues is that they can offer ***at-most-once*** delivery and ***at-least-once*** delivery but never exactly once, in comparison to Kafka that offers ***exactly-once*** [34].

RabbitMQ is also focused on optimizing near-empty queues or empty queues, that is because as soon as an empty queue receives a message, the message goes directly to a consumer. In the case of non-empty queues the messages has to be enqueued and dequeued which in turn results in a slower overall message processing.

Chapter 3

Pre-work

The area of testing message brokers has a wide range of published materials ranging from white papers to blog entries. The amount of information from which one can read and gain knowledge is therefore plenty but the most important aspect is how relevant it is to the work being conducted. In this thesis project the quantative and ambigous variable enqueueing performance is analyzed and evaluated.

Reading the related work and too see how others have tested the message brokers resulted in finding one related paper conducted in September 2017 by Dobbelaera et. al [35]. This paper is examined in its essence in section 3.3.

3.1 Metrics

Enqueueing performance focused on in this thesis can be measured in two ways, **throughput** vs **latency**. The throughput in this case is measured with how many messages that can be sent with focus on how to maximize it. Latency on the other hand can be measured in two different ways, either as one-way (end-to-end) or the round-trip time. For this thesis only the one-way latency is measured. This was done to see the impact during the experiments when tuning the parameters for how long each consumer has to wait before processing an event or message.

Furthermore because these two message brokers have their own unique architecture and design goals when it comes to how to they send and store messages, two other secondary metrics to measure was chosen, the resource usage of the CPU and memory. The CPU metric measured how much of the CPU is allocated to system and to the user while the memory metric focused on how much data is actually written to the harddrives. These two metrics was chosen over others such as the protocol overhead created when sending messages because it was deemed unfeasible to keep statistics of each message being sent over the network link and because

RabbitMQ employs different protocols in comparison to Kafka which has its own binary protocol over TCP.

3.2 Parameters

Because of how Kafka and RabbitMQ are designed the parameter sets between them both are not identical, that is, that there is no 1 to 1 relation where parameter X in Kafka is the same to parameter Y in RabbitMQ. Therefore an investigation of which parameters for both architecture was made. The findings showed that Kafka is left with more configuration parameters than for RabbitMQ and these are described below.

3.2.1 Kafka parameters

- **Batch size** - The batch size is where the producer tries to collect many records in one request when there are many messages being sent to same partition in the broker, instead of sending each message individually. This parameter enhances the performance for both the client and server.

- **Linger** - This parameter works together with the batch size because batching happens during loadtime when there is alot of messages that cannot be sent out faster than they come. When this happens one can add a configurable delay (in milliseconds) for when to send the records instead of trying to send each record as fast as possible. By adding this delay the batch size can fill up and therefore resulting in more messages being sent. If the batch size is reached before the lingering time has finished the former will take precedence over the latter.

- **Acks** - This parameter has three different levels that can be set for the producer. Acks in this case is the number of acknowledgements which the producer is requesting from the leader to have from the replicas before recognizing a request as finished.

  The first level is the most basic one where the leader does not have to wait for an acknowledgement from its followers. This means that a message is considered delivered after it has left the network socket even if there is no acknowledgement from server that it has actually received it.

The second level is where the leader will have the message being written locally and thereafter responding to the producer without waiting for any acknowledgements from the other replicas. This can lead to messages being lost if the leader goes down before the replicas has written it to their local log.

The third level is where the leader will not send an acknowledgement to the producer before receiving all acknowledgements from the replicas. This means that the record can not be lost unless all of the replicas and the leader goes down.

- **Compression** - The compression parameter is used for the data being sent by the producer, Kafka supports three different compression algorithms, **snappy** and **gzip** and **lz4**.
- **Log flush interval messages** - This parameter decides how many messages in a partition should be kept before flushing it to the harddrive.
- **Partitions** - The partitions is how Kafka parallelize the workload in the broker.

3.2.2 RabbitMQ parameters

- **Queues** - This parameter can be configured to tell the broker how many queues should be used.
- **Lazy queue** - The lazy queue parameter changes the way RabbitMQ stores the messages, RabbitMQ will try to deload the RAM usage and instead store the messages to disk when it is possible.
- **Queue length limit** - This parameter can be tuned to either keep track of how many messages there are in the queue or how many bytes are stored.
- **Direct exchange** - The direct exchange parameter is used to declare that the queues being used should be have each message directly sent to its specific queue.
- **Fanout exchange** The fan-out exchange ignores the routing to specific queues and will instead broadcast it to every available queue.

- **Auto-ack** - Auto acknowledgement is used for when the message is considered to be sent directly after leaving the network socket, in similar fashion to the first level of Kafkas acknowledgement.
- **Manual ack** - A manual acknowledgement is when the client sends a response that it has received the message.
- **Persistent** - The persistent flag is used to write messages directly to disk when it enters the queue.

These parameters resulted in working with two different types of subsets which led to the conclusion that RabbitMQ and Kafka can not be compared on equal basis. This made the focus of the thesis work changed to see what degree the different parameters can achieve when it comes to throughput and latency.

3.3 Related work and novelty

Testing of RabbitMQ against Kafka has not been done before except for what was reported in [8] according to my findings. This report focused on two different deliverance semantic the **at-least-once** and **at-most-once** and was tested on RabbitMQ version 3.5 and Kafka version 0.10. Note that version 0.11 and higher for Kafka offers exactly-once and was planned initially in this thesis to have experiments conducted on, but without a counterpart found in RabbitMQ it was left out for testing. This report has its experiments conducted on Kafka version 1.1 and RabbitMQ version 3.7.3.

The experiments found in [8] was conducted on a single bare-metal server and not on a cloud platform which is what this report did and therefore making it more distributed.

Furthermore the experiments conducted for this thesis were tested on multiple servers which is further described in section 4. From [8, p.13] they state that they only used the default configuration for their experiments which is in opposite to this thesis which tests different parameters and their impact on throughput and latency when running in the two different semantics. Additionally the report focuses on the results found by configuring the amounts of partitions, topics, and message size for Kafka and only the message size for RabbitMQ.

The report in question notes that three important factors for measuring throughput is the record size, partition and topic count, albeit true one can read from [36] that there are several other parameters that can factor in the throughput which the authors of [8] does not make use of, which is mainly the *batch size*, *lingerms*, *compression algorithm*, *acks*, and *buffer memory*.

All of these are tested in this thesis except for the buffer memory which is used in combination when you have many partitions. These parameters for Kafka and others for RabbitMQ is presented in section 3.2 and has a further explanation on how they can impact the throughput and latency for the different architectures.

For measuring CPU and memory usage two different programs were used, **mpstat** and **dstat**. The mpstat command displays many different types of information such as I/O utilization, hardware interrupts, software interrupts etcetera, but only two were considered the first was the CPU utilization of the system that occurs on kernel level. The other one was the user level and it shows how much of the CPU is utilized by applications. Dstat was used over other tools such as **vmstat** or **iostat** because of the layout of the output being more understandable as it tracks data in different columns.

Chapter 4

Work

The work is divided in to four sections, section 4.1 presents a testing tool that was developed but was scraped in favour of existing tools from their respective distributions found in RabbitMQ and Kafka. Section 4.2 is where a suitable cloud platform was decided upon and the instances chosen on that platform and the reasons why these instances was chosen and it presents the setup of the testing suite. Section 4.3 presents all the experiments that was made for both Kafka and RabbitMQ. Section 4.4 shows the steps from setting up the servers to conducting the experiments to importing them for data processing.

4.1 Initial process

The testing phase started with a hasty decision of developing a tool for testing out Kafka on their platform. This tool was developed in Java with the purpose of sending arbitrary messages to the broker and measuring the throughput and latency.

The tool consists of five classes, an Emptor and Creator, a MessageObject and lastly a MessageObjectSerializer and Deserializer. The creator can be seen as the publisher and the emptor as the consumer. The class diagram in figure 4.1 shows the classes and their attributes and methods. The tester makes use of its own POJOs to send messages and is therefore bound to using a custom made serializer. Kafka supports other serializers such as Avro, Thrift and ProtoBuf [30] but a custom one was made because of the relative simple messages being sent to the broker.

In order to fetch data from the servers a portbinding had to be made locally to the remote server and there after be listened to via JConsole. The JConsole comes with JDK and can be used to monitor different metrics of Kafka that is connected to the consumer, producer and the broker [37]. Moreover JConsole can be used to log the CPU and memory usage of Java program. The metrics that was logged from JConsole measured the amount of messages it receives per second.

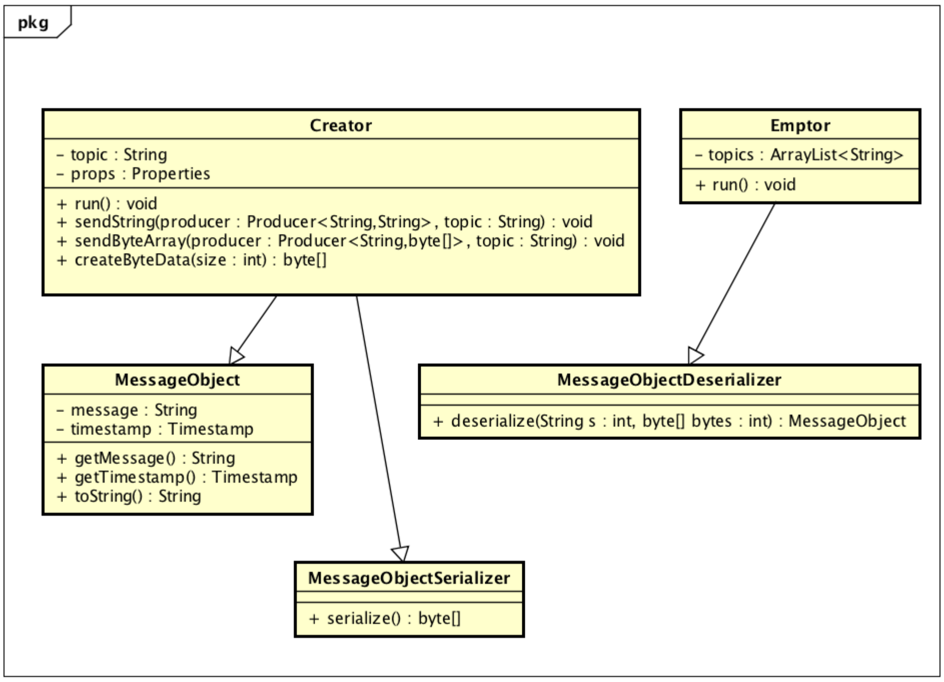

Figure 4.1: Components of Kafka tester

The tool that was created lacked some flexible features such as easily configuring the appropriate parameters of Kafka or adding measurement of multiple servers. In order to test a specific parameter one had to manually change the source code. Furthermore the logging of the CPU usage and memory usage showed unusual findings where the CPU usage was between 90-100%. The experiments conducted ran from a localhost to the cloud platform which could skewer the results since the distance to the cloud platform is longer.

With this in mind a more pragmatic solution was taken where tools provided by Kafka and RabbitMQ was used as they where inherently more flexibile in way where it was possible to connect to multiple servers, track the latency with better statistical measurements and offering more configuration parameters to choose from.

4.2 Setup

One of the key components of this thesis work was choosing a suitable cloud platform to conduct the experiments on. CloudKarafka offers instances provided by Amazon Web Services (AWS) and Google Compute Engine while CloudAMQP offers more in addition to the aforementioned such as DigitalOcean, RackSpace, Azure, SoftLayer and Alibaba Cloud. With only two services in common between the services the choice was left to either Amazon Web Services or Google Compute Engine and the choice fell on AWS.

AWS offers a varity of instances on their platform with focus on different test cases. These are comprised to a general purpose, compute optimization, memory optimization, accerelerated computing or storage optimized [38]. These categories has their own subset of hardware and the first choice to make was to pick which category to create an instance on. Instances on the general purpose section makes use of Burstable Performance Instances which means that if there is a CPU throttle on the host it can momentarily provide additional resources to it. The hardware specifications of these instances can be seen in Table 4.1.

Table 4.1: Instances on general purpose

Model	vCPU	Memory (GiB)	Harddrive storage
t2.nano	1	0.5	EBS-only
t2.micro	1	1	EBS-only
t2.small	**1**	**2**	**EBS-only**
t2.medium	2	4	EBS-only
t2.large	2	8	EBS-only
t2.xlarge	4	16	EBS-only
t2.2xlarge	8	32	EBS-only

EBS-only means that the storage is located on the Elastic Block Store which is a block-level storage that is orchestrated in a way where it replicates a physical storage drive which in comparison to an object store that stores data within a data-object model [39].

Instances found on the compute optimized category can be seen in Table 4.2.

Table 4.2: Instances on compute optimization

Model	vCPU	Memory (GiB)	Harddrive storage
c5.large	2	4	EBS-only
c5.xlarge	4	8	EBS-only
c5.2xlarge	8	16	EBS-only
c5.4xlarge	16	32	EBS-only
c5.9xlarge	36	72	EBS-only
c5.18xlarge	72	144	EBS-only

The instance chosen to conduct the experiments on was the t2.small because optimizing for throughput and latency for low hardware specifications from an economical perspective is more sustainable for users.

Four instances each of these where created for Kafka and RabbitMQ in total eight instances. Three of each instance where to be tested against and the fourth one acted as a testrunner in order to minimize external factors affecting the results as to testing it locally which was done with the self-made tool. The setup for Kafka and RabbitMQ can be seen in figure 4.2.

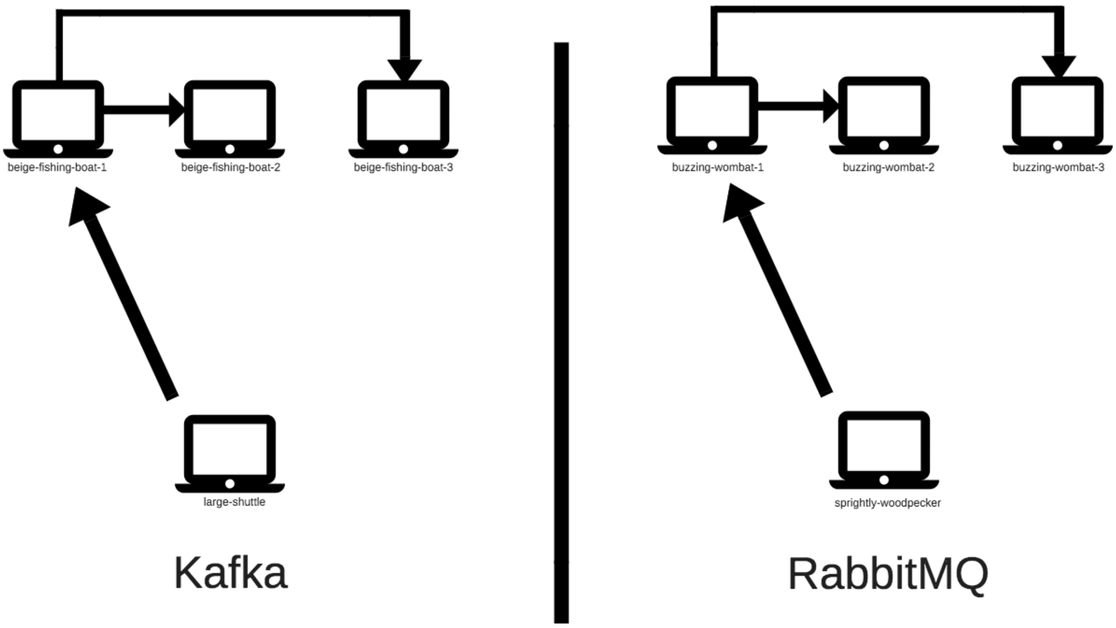

Figure 4.2: Test setup

From figure 4.2 the large-shuttle server acts as a tester for Kafka and sprightly-woodpecker for RabbitMQ, these arbitrary names was given by the respective service. The instance beige-fishing-boat-1 is the main broker and beige-fishing-boat-2 and 3 are replicas. For RabbitMQ buzzing-wombat-1 is the main server and buzzing-wombat-2 and 3 are replicas.

4.3 Experiments

The experiments that were conducted on RabbitMQ and Kafka is presented in section 4.3.1 and 4.3.2 . The messages size was set to 500 bytes.

4.3.1 Kafka

Experiment one

The first experiment for Kafka was to configure the batch size and then trying it out with a number of partitions. The batch size was set to an interval from 10 000 to 200 000 with a step size of 10 000. The number of partitions were set to start with 5 and then iteratively being changed to 15, 30 and lastly 50. This experiment ran with a snappy compression. The number of acknowledgements was set to level one for one set of experiments and level three for the other set.

Experiment two

The second experiment tested the linger parameter in combination with batch size. The batch size was set to 10 000 to 200 000 with a step size of 10 000. The linger parameter had an interval between 10 to 100 with a step size of 10. For each batch size the linger parameter was tested, that is, for batch size 10 000 the linger parameter interval tested was 10 to 100, for batch size 20 000 the linger parameter set was 10 to 100 and so forth. The compression for this experiment was snappy. Because of this extensive testing only 2 partitions was used, 5 and 15. The acknowledgement level was set to one for one set and level three for the other set.

Experiment three

The third experiment focused on the impact of compressing the messages, this experiment tested firstly a batch size interval of 10 000 to 200 000 with acknowledgement level two and snappy compression. The second experiment tested the same but without any compression at all.

Experiment four

The fourth experiment tested flush interval message parameter in order to see how it affected the disk usage. This experiment was conduected with two different partions, 5 and 15. The flush interval message parameter was tested in an interval as such 1, 1000, 5000, 10 000, 20 000, 30 000. The compression used was snappy.

4.3.2 RabbitMQ

Experiment one

This experiment tested the fanout exchange, with queues from 1 to 50. This test was conducted firstly with manual acknowledgement and thereafter auto acks. This was tested with persistent mode.

Experiment two

This experiment tested lazy queues with queues from 1 to 50, this was conducted with manual and auto acknowledgements. This was tested with persistent mode.

Experiment three

The third experiments tested five different queue lengths ranging from the default length to 10, 100, 1000 and 10 000. This was tested with manual and auto acknowledgements and with the persistent mode.

4.4 Pipelining

The experiments was deemed to be unfeasible to conduct by manually changing the parameters for each experiment, this issue was solved by developing a script that can run each experiment automatically and log the findings.

This script works as such; it will connect to beige-fishing-boat or buzzing-wombat and start logging with dstat and mpstat to two text-files, after this, it connects to either the large-shuttle server or sprightly-woodpecker depending on which message broker to be tested and start sending messages to the message brokers and logging it to a text-file.

After this step another connection to the beige-fishing-boat/buzzing-wombat needed to be done in order to terminate the processess that is logging with mpstat and dstat in order to not stack multiple process of the same function which could affect the result of the CPU and memory management.

With all of these tests being conducted a large amount of text-files was being generated and it needed to be processed in an efficient way. Therefore a parser was developed that could take in the data from these text-files and generate an Excel sheet instead of manually filling each value from the text-file. This parser was put in front of the brokers in order for it to work, the whole architecture of the script and the servers and the parser can be seen in figure 4.3 with the appropriate steps on the whole run.

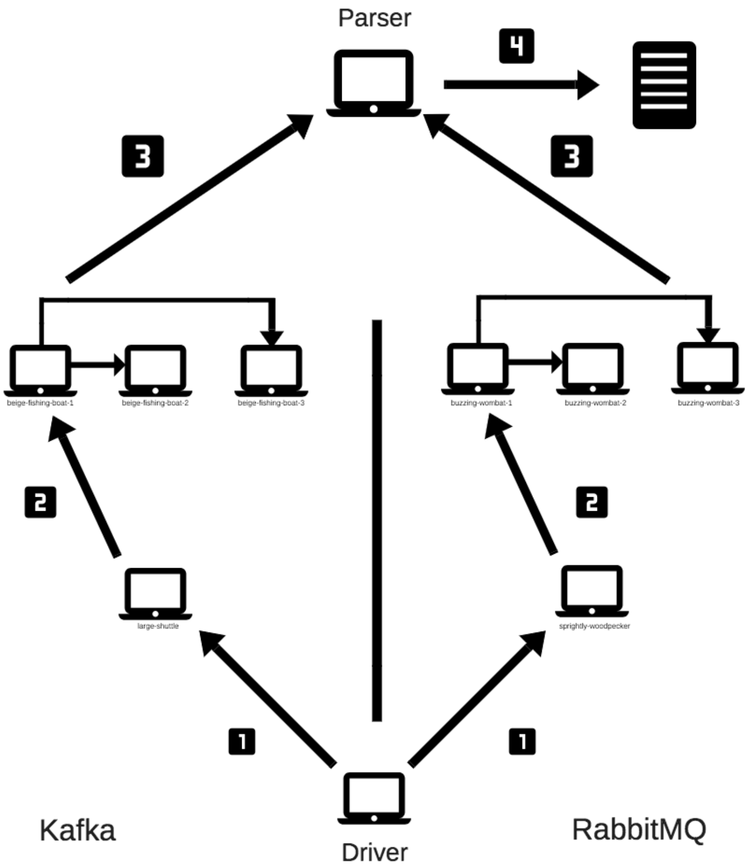

Figure 4.3: Architecture of system

Chapter 5

Result

A subset of results are presented in section 5.1 and 5.2, these results are chosen out of many because they show the most apparent difference between the experiments. Other results are found in the appropriate appendix.

5.1 Kafka

5.1.1 Experiment one

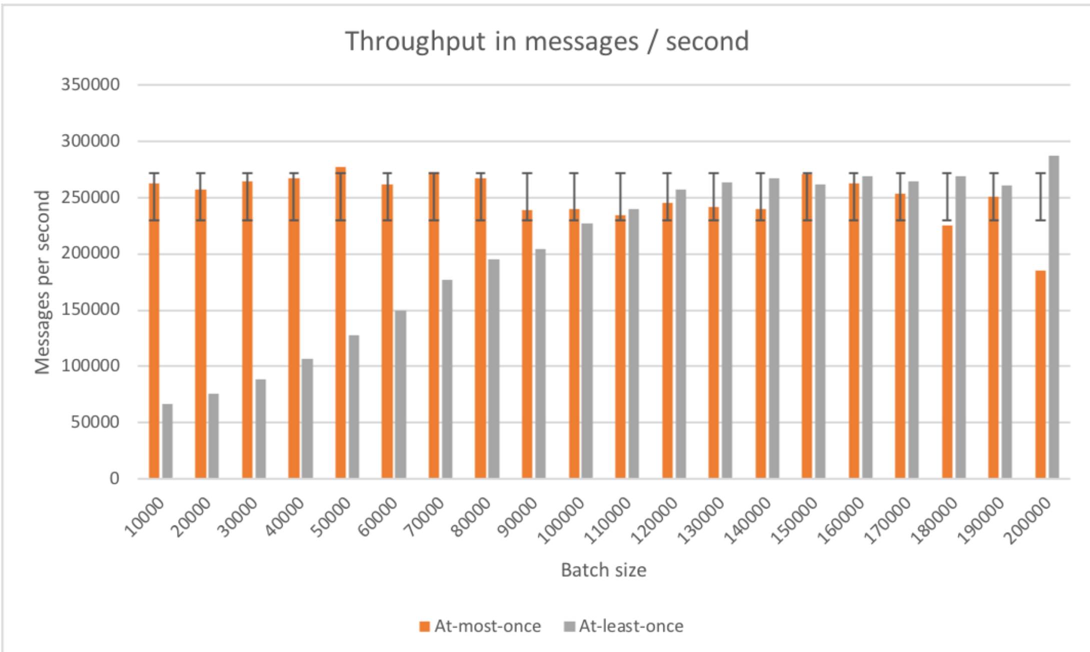

Figure 5.1: Throughput measured with 5 partitions

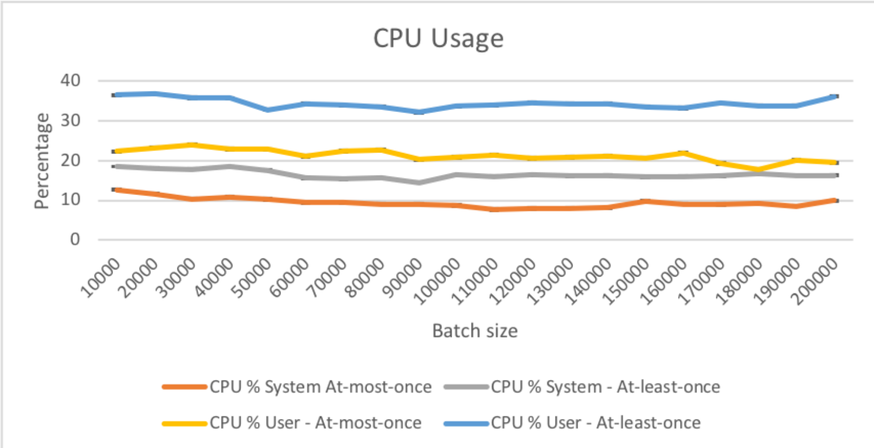

Figure 5.2: CPU Usage with 5 partitions

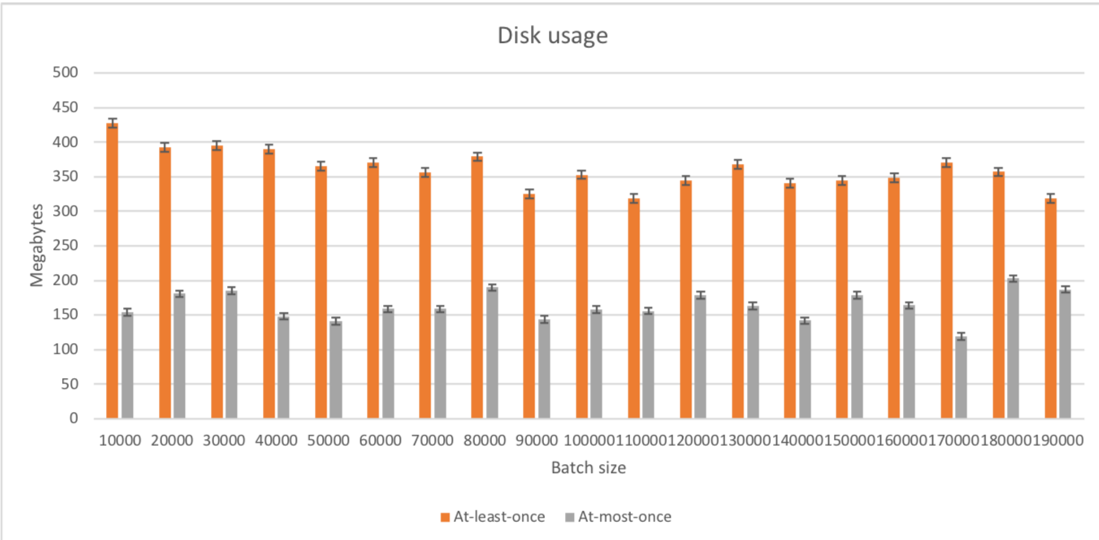

Figure 5.3: Diskusage with 5 partitions

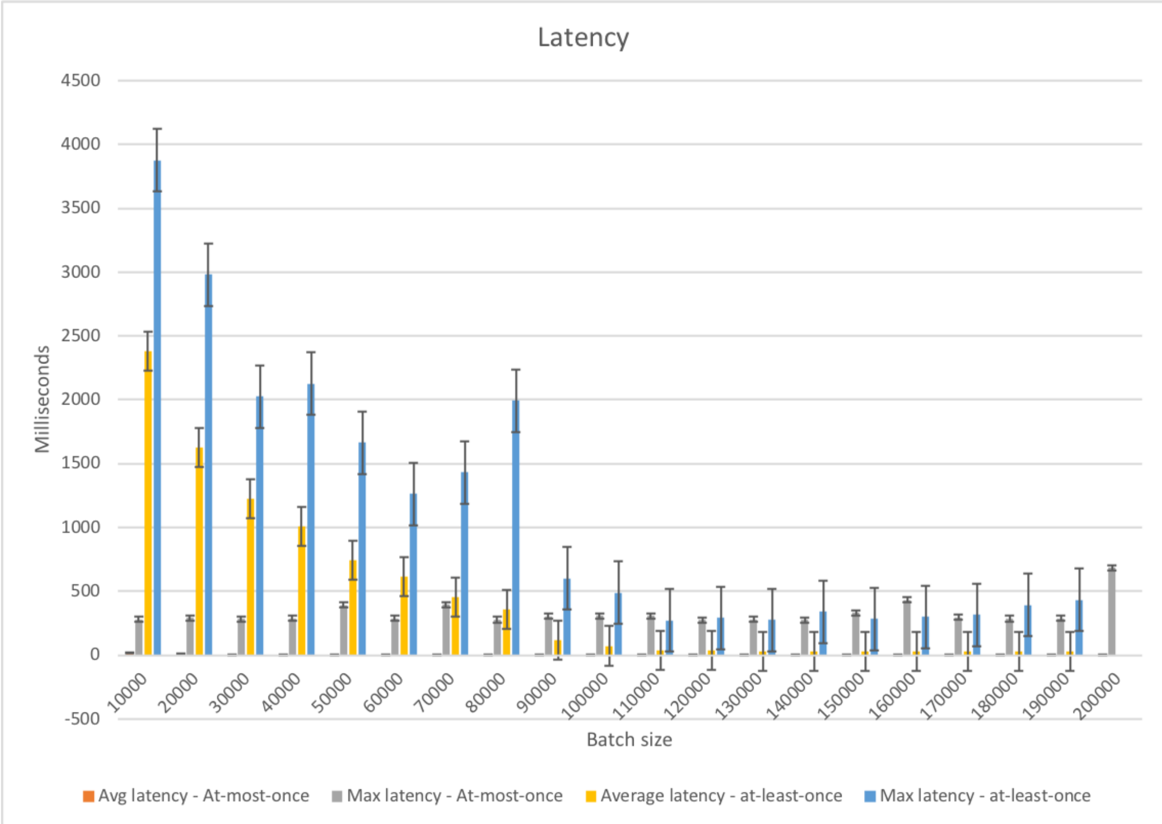

Figure 5.4: Latency measured with 5 partitions

5.1.2 Experiment two

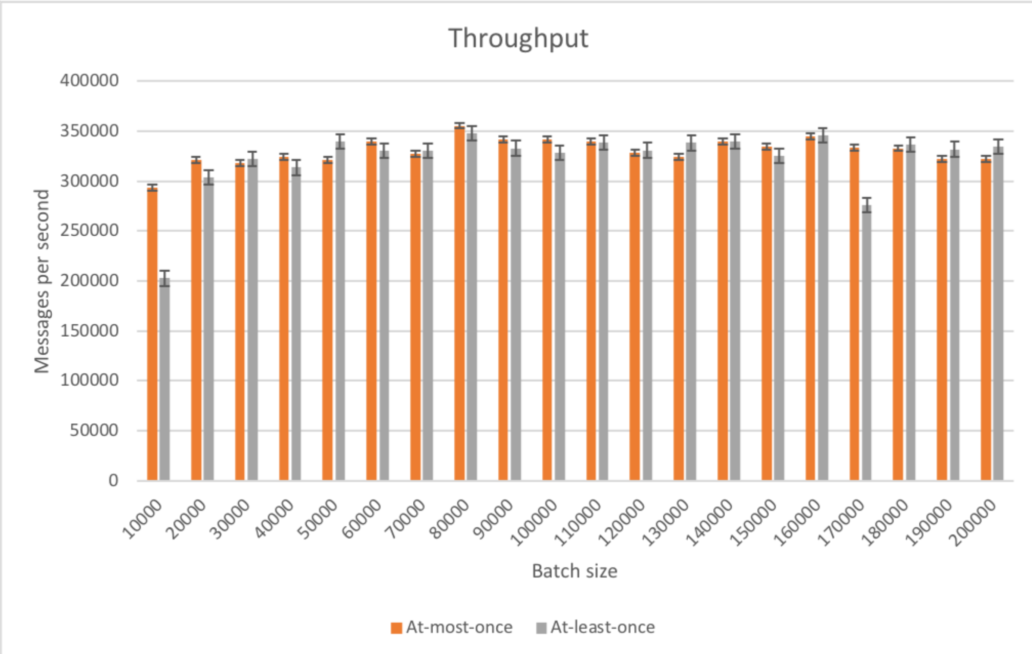

Figure 5.5: Throughput with linger set to 10

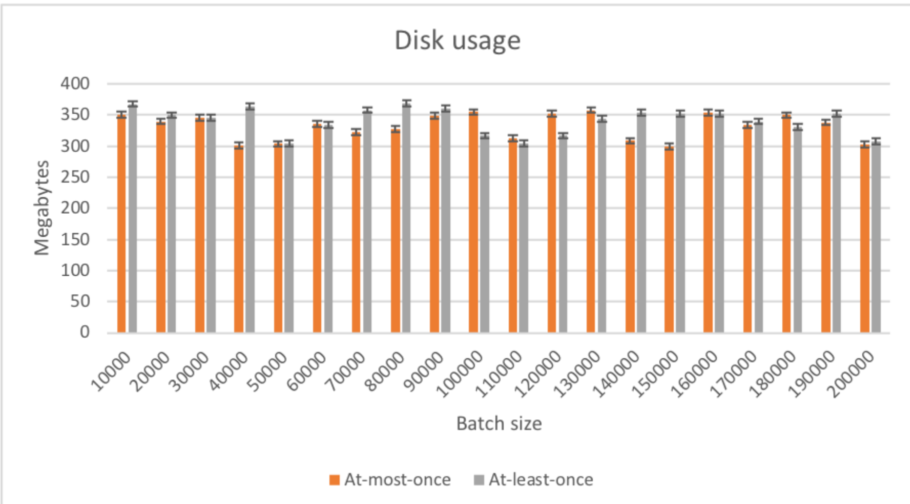

Figure 5.6: Disk usage with linger set to 10

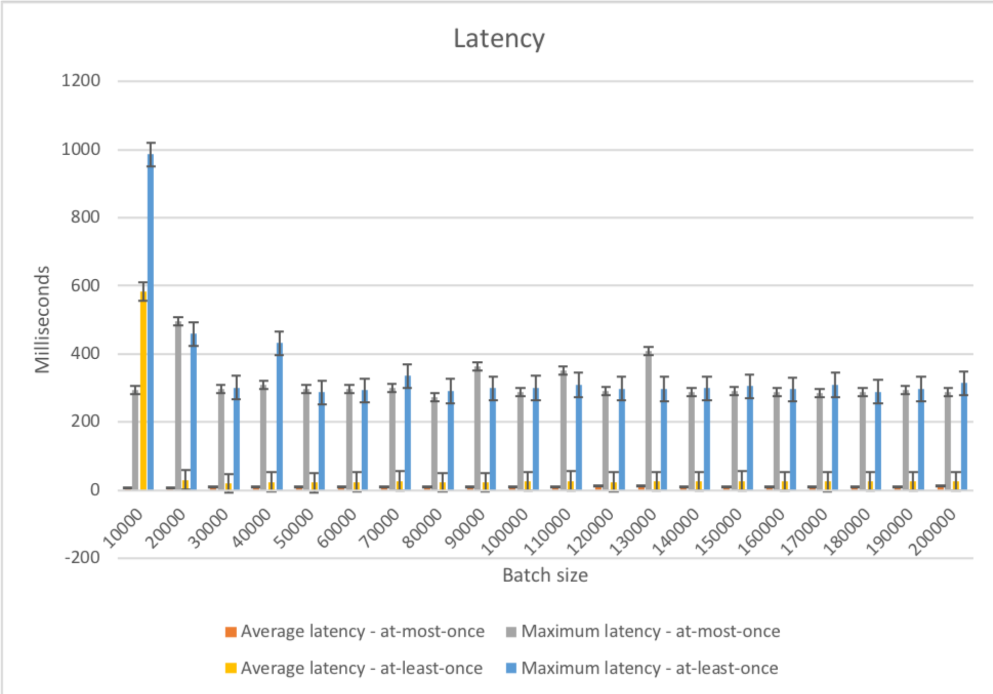

Figure 5.7: Latency with linger set as 10

5.1.3 Experiment three

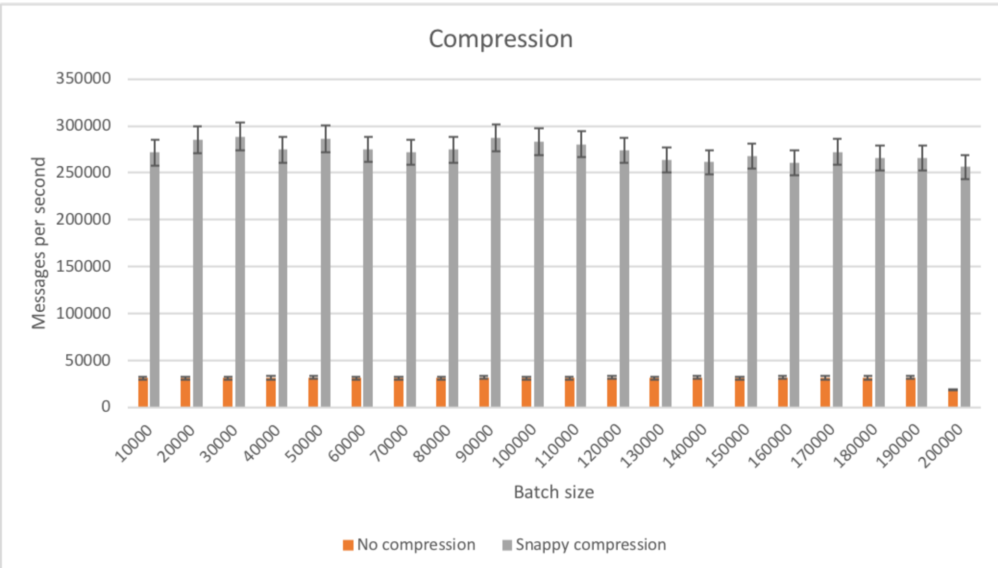

Figure 5.8: Throughput measurements with snappy compression and none compression

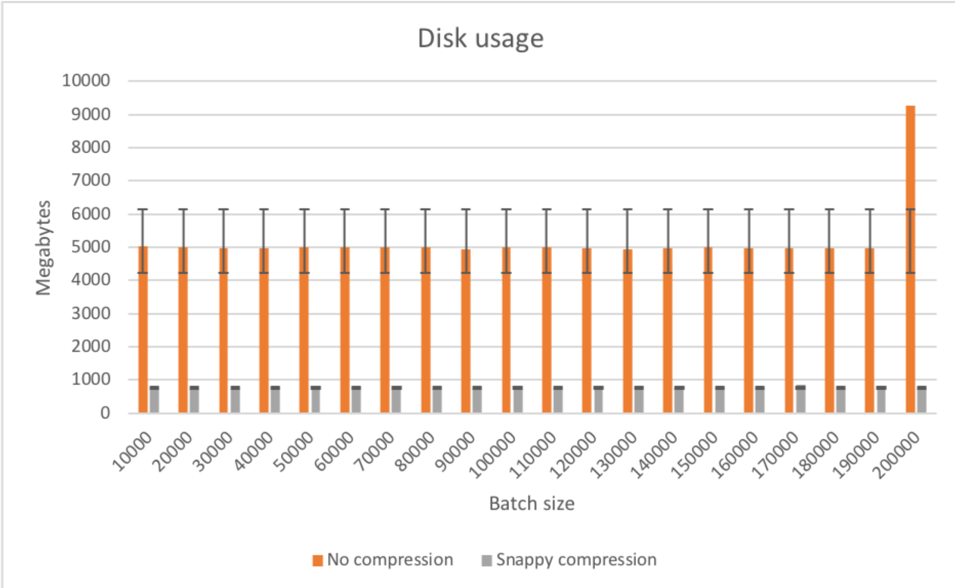

Figure 5.9: Disk usage with compression and none compression

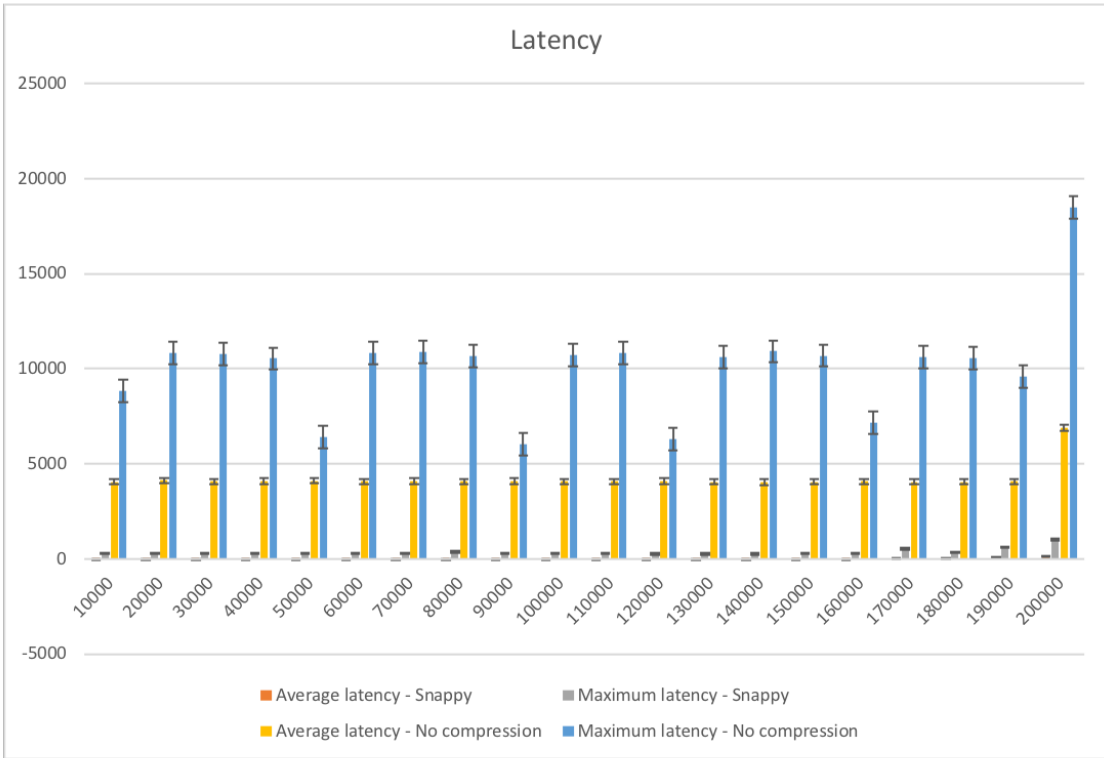

Figure 5.10: Latency measurements with compression and none compression

5.1.4 Experiment four

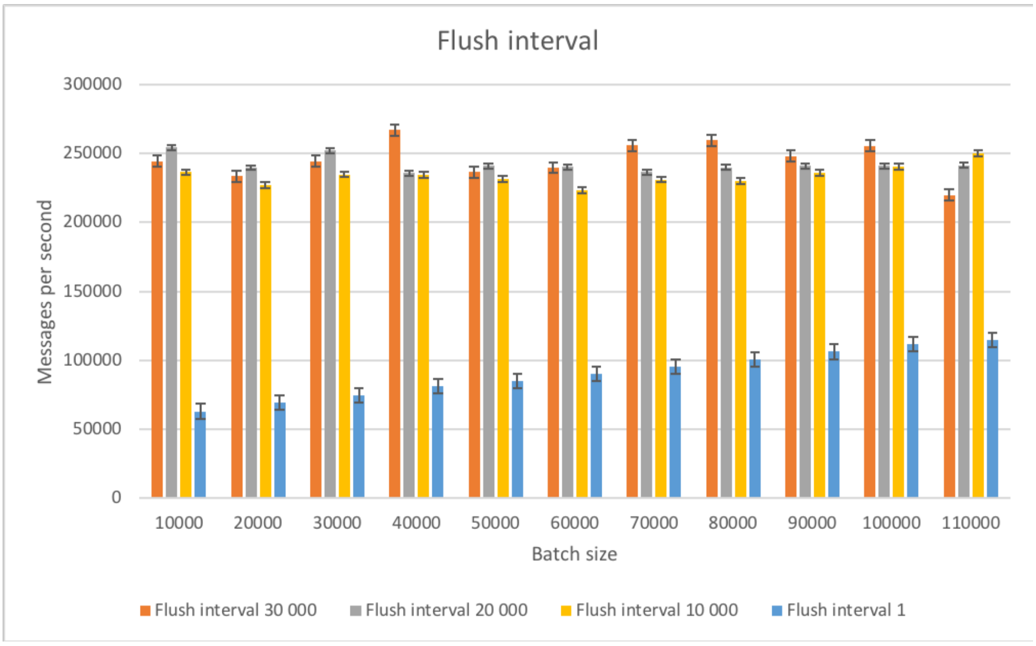

Figure 5.11: Throughput with configured flush interval

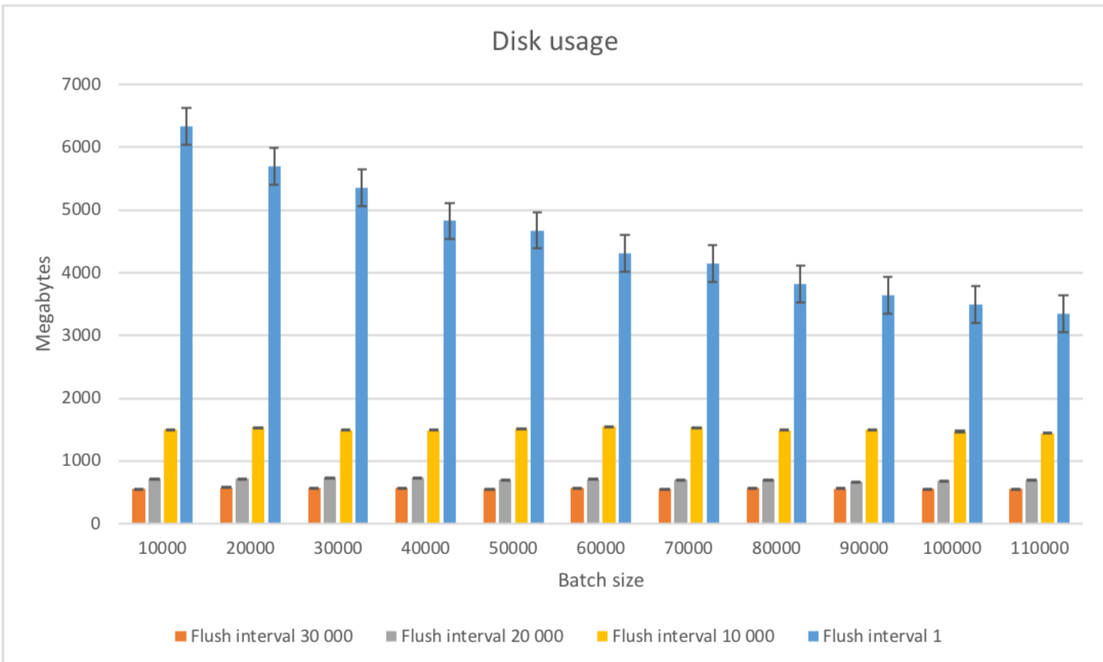

Figure 5.12: Disk usage with configured flush interval

5.2 RabbitMQ

5.2.1 Experiment one

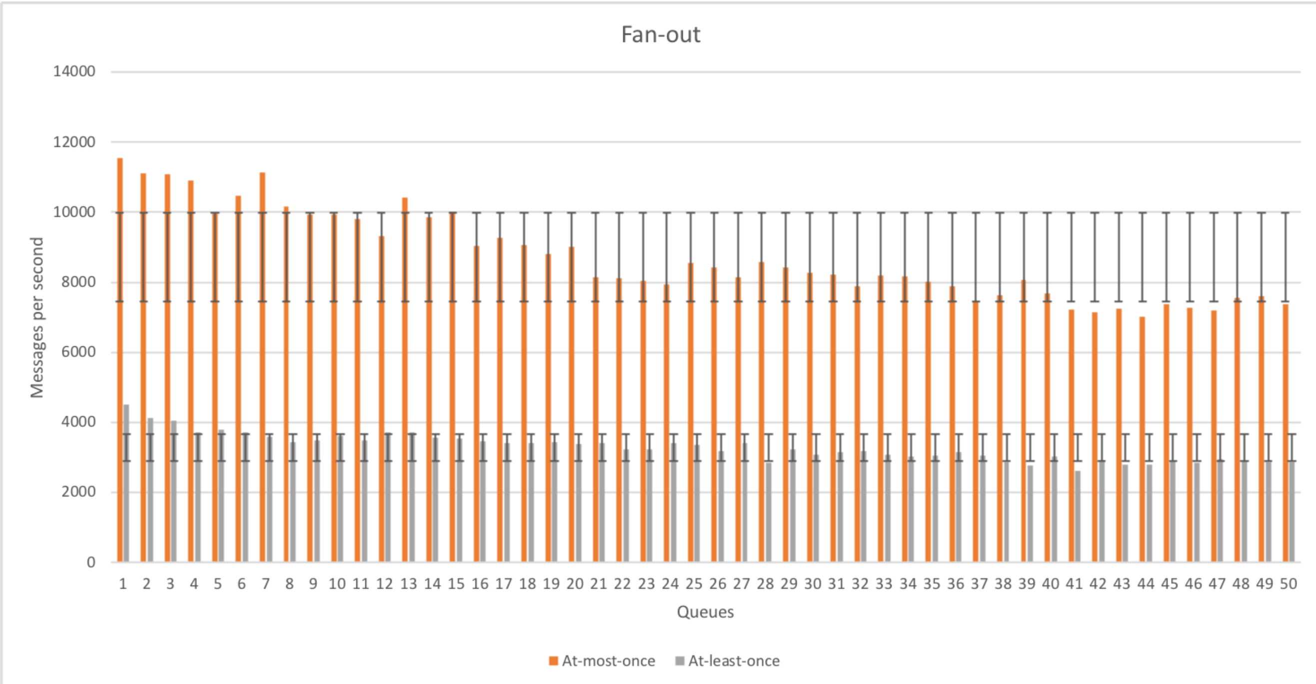

Figure 5.13: Throughput with fanout exchange

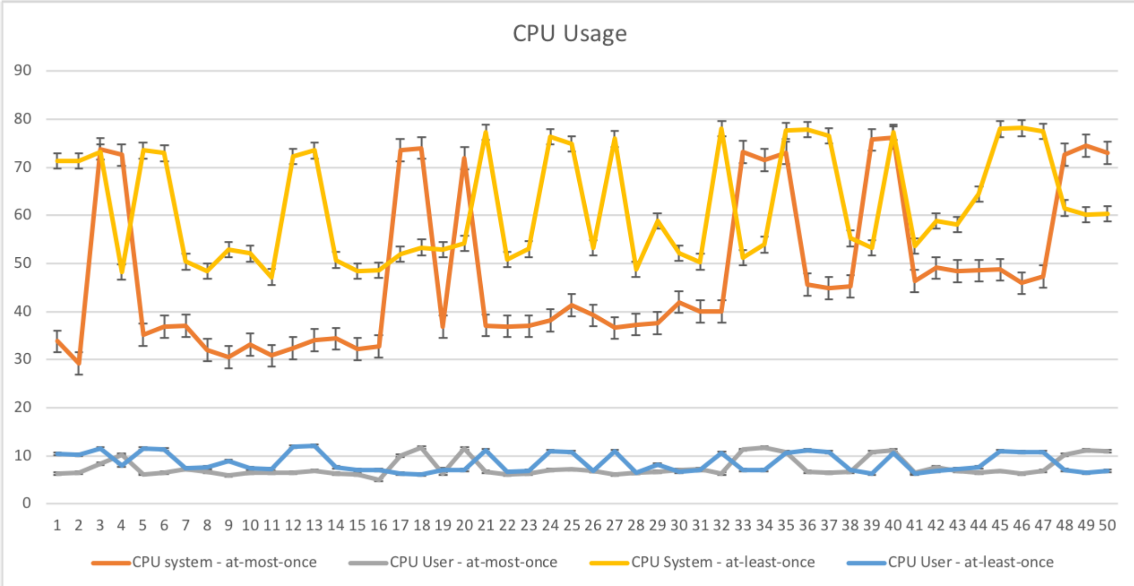

Figure 5.14: CPU usage with fanout exchange

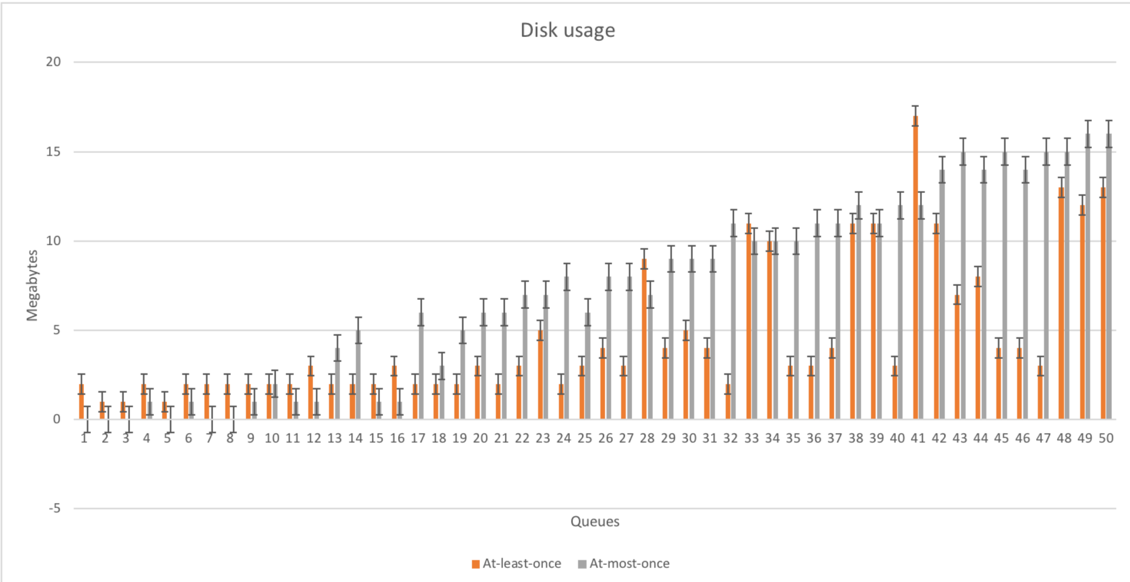

Figure 5.15: Disk usage with fanout exchange

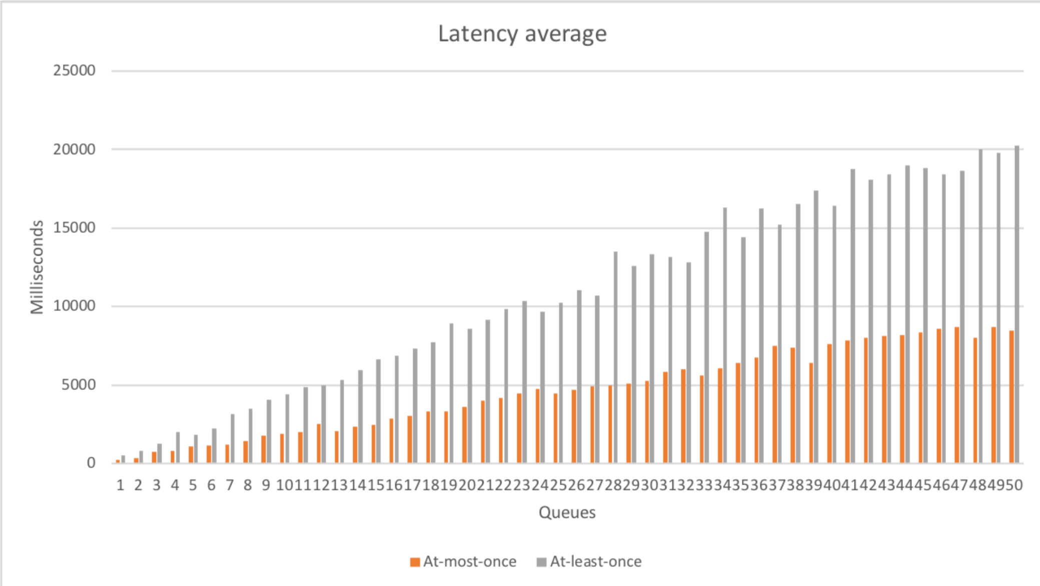

Figure 5.16: Latency measured with fanout exchange

5.2.2 Experiment two & three

The results from experiment two and three shows similar results to in both at-least-once and at-most-once mode to what is presented in experiment one and can be seen in the appendix section A.2 as rawdata. This is discussed in the conclusion section on to why it could be like that.

Chapter 6

Conclusion

A number of parameters for Kafka and RabbitMQ has been tested in this thesis in two different delivery modes, at-least-once and at-most-once. The goals presented in 1.4 has been met. One of the drawbacks of the thesis is the available parameter sets for RabbitMQ which is not that versatile in comparison to Kafka which hindered some experiments to be made since the equivalent part was not found in RabbitMQ.

When it comes to the result of the experiments one has to start with Kafka. This broker performs much better than RabbitMQ according to the tests when it comes to throughput and latency. The amount of messages being sent with Kafka is over 5 times more than with RabbitMQ. This can be seen in comparison to figure 5.13 and figure 5.1 with RabbitMQ reaching 12000 messages and Kafka reaching over 250 000.

Another interesting point to see is that impact of applying compression with Kafka, with compression Kafka reaches over 250 000 messages per second but without it, it only goes up to approximately 50 000 which can be seen in figure 5.8. Furthermore the disk usage when applying compression and non-compression is staggering, with compression it only writes 1Gb to the disk and without its almost 5 times more which can be seen in figure 5.9.

The linger parameter for Kafka combined with batch size gave the throughput an extra boost in comparison to not having it configured, by almost 50 000 more messages than by just using the batch size parameter, this can be seen in 5.7 which has over 300 000 messages being sent for the majority of batches.

The flush interval parameter for Kafka hampers the throughput when Kafka is forced to flush every message to the disk and it only reaches approximately 50 000 msg/s. With compression it still writes more to the disk (between 7Gb to 4Gb depending on what batch size is configured) in comparison to not applying compression which can be seen in figure 5.12.

Now as to why the experiments for RabbitMQ performed so poorly in comparison to Kafka is that the experiments conducted with many different queues only were configured to having 1 consumer/producer using them which is the default mode of the Kafka tool.

Another explanation to why Kafka outperforms RabbitMQ is that Kafka is optimized for stream-based data and does not offer the same configuring options as RabbitMQ with the potential workings of an exchange and binding messages with keys. RabbitMQ is more versatile when you want to configure specific messages to go to certain consumers.

The CPU utilization for Kafka and RabbitMQ are notably different which one can see from figure A.10 and 5.2. The CPU utilization for RabbitMQ with the at-least-once mode staggers between 30-40% and spikes up to 70%. This behaviour can be explained that during this time that the queue is empty and therafter needs to have it messages stored.

A very notable different between Kafka and RabbitMQ is that Kafka makes use of the disk in terms of how much data is written in comparison to RabbitMQ. From figure 5.15 one can see that RabbitMQ writes maximum of around 20 megabytes in comparison to Kafka which in some experiments showed over 5 Gb of data being written.

Possible future work could be to test it on other hardware configurations and try to set up a larger cluster of nodes. It would also be interesting to see the performance impact when running Kafka in exactly-once mode.

Bibliography

[1] EMC Digital Universe with Research Analysis by IDC. *The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things*. 2014. URL: https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[2] Brendan Burns. *Designing Distributed System. Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly, 2017.

[3] Dotan Nahum Emrah Ayanoglu Yusuf Aytaş. *Mastering RabbitMQ. Master the art of developing message-based applications with RabbitMQ*. Packt Publishing Ltd, 2015.

[4] Gregor Hohpe. *Enterprise integration patterns : designing, building and deploying messaging solutions*. eng. The Addison-Wesley signature series. Boston: Addison-Wesley, 2004. ISBN: 0-321-20068-3.

[5] "Message-Oriented Middleware". In: *Middleware for Communications*. John Wiley Sons, Ltd, 2005, pp. 1–28. ISBN: 9780470862087. DOI: 10.1002/0470862084.ch1. URL: http://dx.doi.org/10.1002/0470862084.ch1.

[6] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683.

[7] Vasileios Karagiannis et al. "A Survey on Application Layer Protocols for the Internet of Things". In: *Transaction on IoT and Cloud Computing* (2015). URL: https://pdfs.semanticscholar.org/ca6c/da8049b037a4a05d27d5be979767a5b802bd.pdf.

[8] Philippe Dobbelaere and Kyumars Sheykh Esmaili. "Kafka Versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. Barcelona, Spain: ACM, 2017, pp. 227–238. ISBN: 978-1-4503-5065-5. DOI: 10.1145/3093742.3093908. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/3093742.3093908.

[9] *CloudAMQP - RabbitMQ as a Service*. URL: https://www.cloudamqp.com/ (visited on 03/08/2018).

[10] *CloudKarafka - Apache Kafka Message streaming as a Service*. URL: https://www.cloudkarafka.com/ (visited on 03/08/2018).

[11] Pieter Humphrey. *Understanding When to use RabbitMQ or Apache Kafka*. 2017. URL: https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka (visited on 03/09/2018).

[12] Farshad Kooti et al. "Portrait of an Online Shopper: Understanding and Predicting Consumer Behavior". In: *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. WSDM '16. San Francisco, California, USA: ACM, 2016, pp. 205–214. ISBN: 978-1-4503-3716-8. DOI: 10.1145/2835776.2835831. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/2835776.2835831.

[13] Ryuichi Yamamoto. "Large-scale Health Information Database and Privacy Protection." In: *Japan Medical Association journal : JMAJ* 59.2-3 (Sept. 2016), pp. 91–109. ISSN: 1346-8650. URL: http://www.ncbi.nlm.nih.gov/pubmed/28299244%20http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5333617.

[14] *Key Changes with the General Data Protection Regulation*. URL: https://www.eugdpr.org/key-changes.html (visited on 03/19/2018).

[15] Bill Weihl et al. "Sustainable Data Centers". In: *XRDS* 17.4 (June 2011), pp. 8–12. ISSN: 1528-4972. DOI: 10.1145/1961678.1961679. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/1961678.1961679.

[16] Anne Håkansson. "Portal of Research Methods and Methodologies for Research Projects and Degree Projects". In: *Computer Engineering, and Applied Computing WORLDCOMP* (2013), pp. 22–25. URL: http://www.diva-portal.org%20http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960.

[17] Jakub Korab. *Understanding Message Brokers*. ISBN: 9781491981535.

[18] Mary Cochran. *Persistence vs. Durability in Messaging. Do you know the difference? - RHD Blog*. 2016. URL: https://developers.redhat.com/blog/2016/08/10/persistence-vs-durability-in-messaging/ (visited on 03/23/2018).

[19] PTh Eugster et al. "The Many Faces of Publish/Subscribe". In: (). URL: http://members.unine.ch/pascal.felber/publications/CS-03.pdf.

[20] Michele Albano et al. "Message-oriented middleware for smart grids". In: (2014). DOI: 10.1016/j.csi.2014.08.002. URL: https://ac-els-cdn-com.focus.lib.kth.se/S0920548914000804/1-s2.0-S0920548914000804-main.pdf?%7B%5C_%7Dtid=76c79d76-8189-4398-8dc8-dda1e2a927e7%7B%5C&%7Dacdnat=1522229547%7B%5C_%7D.

[21] Joshua Kramer. *Advanced Message Queuing Protocol (AMQP)*. URL: http://delivery.acm.org.focus.lib.kth.se/10.1145/1660000/1653250/10379.html?ip=130.237.29.138%7B%5C&%7Did=1653250%7B%5C&%7Dacc=ACTIVE%20SERVICE%7B%5C&%7Dkey=74F7687761D7AE37.E53E9A92DC589BF3.4D4702B0C3E38B35.4D4702B0C3E38B35%7B%5C&%7D%7B%5C_%7D%7B%5C_%7Dacm%7B%5C_%7D%7B%5C_%7D=1522311187%7B%5C_%7D (visited on 03/29/2018).

[22] Piper Andy. *Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP - VMware vFabric Blog - VMware Blogs*. 2013. URL: https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html (visited on 03/29/2018).

[23] Carl Trieloff et al. "AMQP Advanced Message Queuing Protocol Protocol Specification License". In: (). URL: https://www.rabbitmq.com/resources/specs/amqp0-8.pdf.

[24] Emrah Ayanoglu, Yusuf Aytas, and Dotan Nahum. *Mastering RabbitMQ*. 2015, pp. 1–262. ISBN: 9781783981526.

[25] P Saint-Andre, K Smith, and R Tronçon. *XMPP: The Definitive Guide Building Real-Time Applications with Jabber Technologies*. 2009, p. 310. ISBN: 9780596521264.

[26] *Stomp specifications*. URL: http://stomp.github.io/stomp-specification-1.1.html%7B%5C#%7DDesign%7B%5C_%7DPhilosophy (visited on 04/03/2018).

[27] "Piper,Diaz". In: (2011). URL: https://www.ibm.com/podcasts/software/websphere/connectivity/piper%7B%5C_%7Ddiaz%7B%5C_%7Dnipper%7B%5C_%7Dmq%7B%5C_%7Dtt%7B%5C_%7D11182011.pdf.

[28] Margaret Rouse. *What is MQTT (MQ Telemetry Transport)? - Definition from WhatIs.com*. 2018. URL: https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport (visited on 04/03/2018).

[29] Todd Ouska. *Transport-level security tradeoffs using MQTT - IoT Design*. 2016. URL: http://iotdesign.embedded-computing.com/guest-blogs/transport-level-security-tradeoffs-using-mqtt/ (visited on 04/03/2018).

[30] N. Narkhede, G. Shapira, and T. Palino. *Kafka: The Definitive Guide : Real-time Data and Stream Processing at Scale*. O'Reilly Media, 2017. ISBN: 9781491936160. URL: https://books.google.se/books?id=qIjQjgEACAAJ.

[31] Flavio Junquera and Benjamin Reed. *Zookeeper - Distributed Process Coordination*. 2013, p. 238. URL: https://t.hao0.me/files/zookeeper.pdf.

[32] Brian Storti. *The actor model in 10 minutes*. 2015. URL: https://www.brianstorti.com/the-actor-model/ (visited on 04/08/2018).

[33] *RabbitMQ - Protocol Extensions*. URL: http://www.rabbitmq.com/extensions.html (visited on 04/08/2018).

[34] Neha Narkhede. *Exactly-once Semantics is Possible: Here's How Apache Kafka Does it*. 2017. URL: https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/ (visited on 04/08/2018).

[35] Philippe Dobbelaere and Kyumars Sheykh Esmaili. "Kafka versus RabbitMQ". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17* August (2017), pp. 227–238. DOI: 10.1145/3093742.3093908. arXiv: arXiv:1709.00333v1. URL: http://dl.acm.org/citation.cfm?doid=3093742.3093908.

[36] Confluence.io. "Optimizing Your Apache Kafka TM Deployment, Levers for Throughput, Latency, Durability, and Availability". In: (), p. 2017.

[37] Evan Mouzakitis. *Collecting Kafka performance metrics*. URL: https://www.datadoghq.com/blog/collecting-kafka-performance-metrics/ (visited on 06/15/2018).

[38] *Amazon EC2 Instance Types – Amazon Web Services (AWS)*. URL: https://aws.amazon.com/ec2/instance-types/ (visited on 06/17/2018).

[39] *Amazon EC2 Instance Types – Amazon Web Services (AWS)*. URL: https://aws.amazon.com/ec2/instance-types/ (visited on 06/17/2018).

Chapter A

Appendix

A.1 Experiment two - RabbitMQ

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	11194	137.2	188.6	207.26666	226.06666	0	33,73529	5,724118
2	11280	280.06666	411.86666	461.2	528.26666	0	31,07	5,823333
3	10864	381.25	527.4375	572.375	653.5	0	31,91889	6,953333
4	10721	507.5625	724.5	770.625	883.5625	1	45,34333	6,674444
5	10539	683.3125	931.875	1053.5625	1174.3125	0	72,37944	9,074444
6	10663	699.6875	1177.75	1316.25	1576.375	0	70,26474	9,091579
7	10071	746.8125	1137.3125	1326.9375	1450.75	1	69,61	10,17526
8	10106	973.88235	1462.3529	1585.0	1845.1764	1	32,83368	5,77
9	10383	1099.2941	1637.7647	1724.4117	1993.1176	1	33,72	6,498421
10	10250	1160.1764	1721.1764	1936.2941	2472.0	0	33,75263	7,184211
11	9921	1362.0588	1823.1176	2135.8235	2413.8235	1	33,2525	6,1425
12	9923	1430.6666	1947.7777	2169.8333	3121.0	1	32,769	7,4185
13	9154	1639.1666	2300.5555	2645.7777	5569.2777	4	32,732	6,1755
14	9723	1679.1111	2382.6666	2724.3888	2937.6111	1	35,2835	6,288
15	9537	1596.3333	2554.4444	2925.4444	3368.6666	1	33,4135	5,445
16	9553	1813.8421	2724.6842	3190.0526	6084.4736	0	31,6281	5,212381
17	8981	2089.4210	2998.8421	3467.2631	3791.9473	5	33,64	6,011905
18	9279	2039.2631	3079.4210	3207.9473	4450.1578	0	32,83905	5,708095
19	8447	2381.1666	3729.1666	4398.0	5298.5555	4	32,80818	5,915909
20	8636	2472.6315	3547.7368	3950.3157	4307.5263	7	38,68364	6,309545
21	8586	2602.5555	3986.3888	4112.0	4781.2777	8	38,24818	6,54
22	8467	2652.7368	4082.8421	4627.1052	5193.7368	6	37,06591	6,277273
23	8298	2928.7	4408.9	4979.0	9154.4	7	38,42391	6,23913
24	8342	3260.1052	4640.7894	5355.4210	7007.2631	8	38,97348	6,149565
25	8582	2926.3157	4445.7368	4744.4210	5829.5263	8	73,99348	10,90739
26	8768	3012.95	4671.6	5161.0	8796.4	3	39,1792	6,4944
27	8519	3129.15	4788.45	5001.2	5901.55	8	74,15087	10,81739
28	8423	3243.9473	4903.8421	5096.1052	5889.2105	8	73,1013	11,7787
29	8300	3509.35	5260.4	5583.15	6264.9	10	41,04625	7,035833
30	8362	3551.6	5205.0	5695.3	6758.95	9	69,482	10,0888
31	8136	4044.95	5742.55	6247.2	7416.95	10	73,91042	11,21875
32	7892	3996.1578	5974.8947	6289.7368	7161.4736	12	42,065	6,480833
33	8176	3998.3684	5971.5263	6272.2105	7472.6842	10	74,735	11,52833
34	8151	4258.9	6424.75	6753.65	8882.3	11	44,3856	6,4656
35	8073	3991.8421	6315.5789	6677.1578	7523.7894	11	48,05667	6,67625
36	7864	4467.75	6436.6	6908.45	9090.7	11	44,93885	6,788846
37	7696	4849.45	6633.15	7380.0	8867.05	11	43,93	5,8452
38	7843	4705.4	6964.6	7430.2	8675.35	11	44,1052	6,0288
39	7543	5478.9047	7641.4761	8063.8095	9799.0	13	44,36423	6,22
40	7866	4690.4090	6921.8636	7468.6818	8673.0909	9	47,19963	6,449259
41	7438	5498.0952	7858.5238	8484.7619	10438.523	13	47,94519	6,417778
42	7227	5672.6956	8026.6521	8692.3478	9966.8260	13	49,11714	6,356786
43	8794	4844.6956	6758.0	7350.8260	8851.3478	2	73,14481	10,79667
44	7692	5635.1818	7769.5	8504.9090	9409.3636	14	74,96333	10,44519
45	7383	5238.7619	7685.0952	8588.2857	9331.0476	15	73,61444	10,85
46	7392	5793.8260	8241.1739	8922.3478	10895.043	14	76,29786	10,25643
47	7489	6088.6363	7965.2727	8991.3636	10425.227	15	75,31111	11,04852
48	7590	5839.9545	8235.7727	9031.7272	10612.318	15	75,48536	10,31036
49	6948	6789.3043	9670.9130	10019.695	11230.0	15	48,23069	7,325172
50	7589	6453.3043	8792.0	9921.1304	11826.782	16	49,84793	7,598621

Figure A.1: Lazy queue in at-most-once

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	4364	336.0625	444.0	495.25	534.9375	2	50,205	8,406667
2	4072	731.1875	1034.1875	1115.1875	1260.625	2	54,83333	7,875556
3	4117	779.375	1142.375	1282.9375	1443.0	2	72,60474	10,58842
4	3795	1119.7058	1532.1176	1811.9411	1919.5294	5	49,55579	9,028947
5	3677	1498.4444	2294.1666	2673.5	3472.5	1	51,6865	6,946
6	3662	1838.0555	2729.1666	2928.2222	3386.2222	2	50,1681	7,023333
7	3671	2120.7894	3103.0526	3407.2631	3918.7894	2	73,94714	12,22905
8	3544	2338.2105	3269.3684	3753.5263	3934.9473	2	50,37476	7,95619
9	3472	2687.0	3807.25	4302.75	4883.25	2	50,04409	8,051818
10	3417	2895.35	4259.25	4975.2	5669.55	1	51,8213	8,192609
11	3382	3744.2727	5281.9090	5741.1818	6866.0909	2	54,11	9,66875
12	3600	4071.5454	5346.5	5893.5454	7325.5	2	50,66208	7,87625
13	3646	4292.7727	5781.0454	6317.0909	7219.1363	2	49,942	7,2576
14	3571	4881.0833	6370.6666	7079.4166	8295.25	2	49,74423	6,711538
15	3395	5201.7826	6898.1739	7879.3478	9262.9130	1	49,82615	6,898077
16	3480	5507.6666	6882.0416	7774.2083	8492.6666	2	50,785	6,279615
17	3423	6323.7307	8131.1538	8727.2307	9311.1538	2	50,31464	6,885714
18	3370	6037.3461	8183.6538	8873.1153	10055.538	2	51,78241	6,495862
19	3355	7087.8076	9471.3461	10165.384	10951.038	2	53,47345	7,482414
20	3372	7034.6538	8803.8461	9636.4230	10369.230	3	52,13828	7,240345
21	3362	7398.9230	9243.5384	9764.6923	10639.115	2	74,977	11,399
22	3357	7438.0	9479.4615	9901.1153	10749.730	3	74,944	11,752
23	3116	8828.0689	11021.448	11778.965	13130.896	3	50,00788	7,2
24	3276	8551.2068	10647.689	11672.689	13059.172	2	55,95875	7,788125
25	3190	10149.875	12914.562	14098.093	15492.437	3	54,56857	7,197429
26	3272	9777.2903	11800.483	12862.903	14017.129	3	54,33647	7,159412
27	3348	8359.5714	10664.428	10980.464	12192.142	2	76,24969	11,60219
28	3389	8629.7931	10872.206	11509.517	12550.344	3	76,5303	10,84727
29	3219	10561.875	13344.125	14158.156	14997.718	5	55,27444	7,540278
30	3138	10040.193	12544.612	13041.709	14672.258	4	52,31886	7,532571
31	3104	11933.294	14115.441	15092.117	15833.441	6	50,09789	6,969474
32	3190	10988.562	13081.0	13732.656	14850.625	2	78,44861	11,43861
33	3087	12533.428	14648.114	15750.4	17151.314	10	51,69385	6,488974
34	3261	10833.060	13238.181	13865.606	15965.303	2	77,98324	11,23243
35	3065	12658.388	15376.972	16420.388	17390.805	8	55,02775	7,548
36	3156	12276.142	14903.542	15350.142	17024.628	3	78,2041	11,43205
37	2942	15112.073	18238.097	19536.243	20367.609	10	57,22044	7,539111
38	2979	13957.324	16193.027	17192.459	18049.540	5	51,66561	6,837317
39	2973	13935.486	16444.756	17125.540	18400.702	2	80,24951	10,52341
40	2885	14481.078	17043.921	17612.789	18475.736	3	80,87381	11,47881
41	2834	14947.5	17755.85	18483.175	19564.175	16	56,71591	7,493409
42	2986	13708.342	16713.5	17482.684	18436.210	3	78,68452	12,18452
43	2946	14377.0	17653.358	18271.769	18937.794	4	79,52767	10,37721
44	2676	17376.422	20002.888	21030.755	22096.311	12	59,1056	7,6538
45	2853	18817.583	22112.833	23166.708	23769.375	9	63,34077	8,085769
46	2798	16773.720	19194.441	20140.209	21332.976	4	78,6125	10,69083
47	2872	17246.636	20223.977	21152.977	21848.795	11	61,21184	7,871224
48	2845	17247.477	20116.045	20837.977	21659.75	10	60,49531	7,046531
49	2903	17624.888	20402.555	21105.466	21995.244	14	59,19143	6,059796
50	2909	19704.02	23154.58	24353.9	24926.18	17	58,91926	5,677037

Figure A.2: Lazy queue in at-least-once mode

A.2 Experiment three - RabbitMQ

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	2603	578.93333	861.2	970.13333	1115.2	2	53,13632	8,865263
2	4087	723.5	984.8125	1060.375	1207.1875	1	73,515	10,775
3	3787	843.75	1178.9375	1400.4375	1602.125	1	52,83167	7,752778
4	3655	1154.7058	1623.6470	1890.7647	1955.8823	2	48,78316	8,071579
5	3633	1406.0	1945.8823	2263.5882	2439.6470	2	48,6315	7,401
6	3676	1678.2777	2433.8888	2727.1666	2977.4444	1	74,1905	10,787
7	3560	1941.8421	3017.7368	3337.3157	3844.5263	2	48,93714	7,472857
8	3544	2319.1578	3178.5789	3661.3157	4154.5789	1	73,36524	12,17143
9	3383	2632.0	3835.55	4303.4	4730.55	1	49,26773	8,171364
10	3443	2951.4285	4146.6666	4841.7619	5418.4761	2	50,81783	8,035217
11	3448	3058.3	4347.15	4862.7	5338.45	1	73,21045	12,66455
12	3671	4033.5454	5226.6818	5847.3636	6404.5454	3	49,24875	7,165833
13	3692	4318.0909	5418.3636	6000.0	7455.6363	2	48,8796	6,4892
14	3516	4343.6086	5836.0869	6263.0434	7146.2173	2	49,43154	6,321923
15	3561	4617.0869	6223.6956	6604.5217	7593.6956	2	50,7736	6,9324
16	3490	5582.6521	7157.9130	7606.7826	8182.1739	3	49,77923	6,566923
17	3441	5567.28	7297.92	8196.28	8825.2	3	50,66074	6,918889
18	3425	6110.0833	7793.8333	8244.5833	9333.8333	3	73,41926	12,17593
19	3474	5935.1666	7820.7083	8154.8333	9113.875	2	74,85593	11,20593
20	3327	6775.08	8654.0	9160.4	11212.92	3	76,26429	11,31786
21	3369	7163.0384	9088.8076	9528.8846	11376.461	2	76,11931	11,6669
22	3285	9225.1666	11563.2	12379.0	14447.4	2	52,86667	7,334848
23	3298	7748.0	9712.1851	10281.037	11908.296	2	76,26667	11,71033
24	3285	8133.8620	10871.517	12095.517	13486.965	3	55,37394	7,435152
25	3144	8933.8620	10967.517	11618.620	12596.862	2	55,52781	7,370313
26	3229	9122.1666	11336.6	12433.3	14032.5	4	54,36061	7,120303
27	3321	8901.2666	11536.2	12664.033	14555.666	3	53,16294	6,273824
28	3184	9340.5517	12035.655	12940.103	13865.137	4	52,65364	7,267576
29	3360	9724.9354	11675.806	12388.064	13698.161	3	76,48765	11,09471
30	3171	10442.125	13083.781	13806.343	15001.937	3	51,96083	6,985
31	3154	11119.781	13082.281	13814.468	14807.125	5	51,78056	7,205556
32	3028	11080.062	13380.843	14028.093	15013.437	8	51,37833	6,934722
33	2990	13337.473	16296.342	18186.842	18977.736	11	51,79714	6,829524
34	2991	12073.542	14549.942	15979.114	16917.4	12	53,09897	7,151282
35	3015	12373.428	14880.628	15858.342	16997.4	7	53,79641	6,762308
36	3192	11715.029	13944.5	15007.529	16595.5	3	78,83184	10,62895
37	2611	15787.170	19237.439	19971.0	20380.560	17	52,38933	7,648667
38	2908	14002.973	16771.210	17793.421	18712.763	5	53,98143	6,919762
39	2913	14721.102	17189.615	18143.128	18805.051	3	78,45326	10,55279
40	2770	16698.465	19388.790	20571.093	21138.604	6	57,28234	7,170638
41	2785	18296.914	21460.510	22366.702	23310.106	18	54,70431	7,277843
42	2831	14666.8	17908.55	18625.5	19731.675	10	57,42378	7,507778
43	2789	15999.452	19133.452	19777.833	20842.142	7	56,0463	6,727826
44	2805	15977.097	18635.682	19305.390	20250.878	9	58,505	6,161957
45	2775	17352.022	20074.288	21130.533	22041.311	10	57,42755	6,253061
46	2724	19227.404	21331.851	22236.765	23051.425	20	57,8802	6,49
47	2923	16575.581	19137.0	19863.604	21142.162	13	57,02875	7,34625
48	3004	15043.825	17737.075	18783.6	19534.4	4	79,11068	10,36091
49	2854	18385.297	21717.914	22477.085	23176.042	11	60,62314	7,211176
50	2804	18801.891	21351.347	22009.956	22970.608	16	57,82608	6,931765

Figure A.3: At least once with queue length 10

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	4207	322.2	422.6	465.66666	504.06666	1	49,17389	7,911667
2	3984	647.75	892.9375	971.25	1092.375	1	53,61611	7,436111
3	3613	1126.125	1563.6875	1735.8125	1977.25	1	65,47421	10,80526
4	3650	1218.6470	1837.5882	1959.4705	2293.1764	1	59,86684	10,23895
5	3644	1444.8235	2298.6470	2575.4117	3826.5882	2	71,11476	10,57714
6	3668	1525.7777	2761.7777	3143.6666	3586.8888	2	72,82667	10,69905
7	3533	2073.7894	2942.0	3284.1578	3586.2105	2	49,04238	7,823333
8	3560	2261.2105	3168.5789	3723.8947	4120.4210	1	73,61952	12,55286
9	3404	2592.4	3982.75	4408.7	5007.25	1	49,845	8,163182
10	3443	3140.1428	4241.4761	4769.6190	5391.5714	3	50,03043	7,799565
11	3387	3664.9047	5024.3333	5577.0	6384.8095	2	50,80375	9,154583
12	3681	3775.7272	5280.7272	5755.1363	6510.9545	2	49,60083	6,839167
13	3623	4129.0	5381.6818	6066.0909	6707.0909	4	48,53917	6,055417
14	3543	4459.2608	6181.7826	6957.6521	7474.0	1	49,7456	6,4552
15	3502	5384.125	6916.125	7437.6666	8029.3333	3	47,69462	6,791923
16	3317	5858.6	7762.28	8320.4	9075.28	2	45,48593	6,41963
17	3442	5130.9166	7103.3333	7446.75	8306.0833	2	51,12423	6,168077
18	3386	6738.7142	8345.8214	10077.0	12220.357	3	49,788	6,551333
19	3498	6302.1666	7978.5833	8283.5833	9263.875	3	74,94786	11,0725
20	3404	6654.6	8543.48	8776.0	10028.0	3	76,45964	11,3925
21	3307	7287.4230	9255.0769	9751.6923	10834.384	2	75,02667	11,88933
22	3338	7756.7777	9656.9629	10495.148	12510.0	3	51,29516	7,698065
23	3231	8599.2857	10287.321	11013.714	11690.464	3	50,79125	6,992188
24	3352	7667.2962	9894.3703	10315.259	11573.037	3	75,68484	10,43871
25	3359	8632.8620	10627.896	11382.172	12589.482	3	77,34781	10,75531
26	3193	8630.3103	11360.206	12246.172	13541.551	3	54,60576	7,026061
27	3261	9519.0645	11714.290	12809.161	13570.548	3	53,62735	7,206765
28	3243	8787.9666	11546.666	12365.133	13068.466	6	51,41735	6,962941
29	3258	9418.4	11687.733	12176.6	13408.633	3	77,12	10,92848
30	3090	10927.812	13067.781	13769.843	14847.062	4	48,80806	7,026667
31	2990	11531.125	13609.0	14127.718	15217.531	4	50,89972	6,164722
32	2938	11499.666	13682.818	14790.606	15934.060	11	50,82395	6,602632
33	3077	11508.181	13483.242	14501.060	15276.636	2	76,98405	10,69622
34	3218	11702.571	14678.685	15704.342	17452.142	4	59,61795	7,014615
35	3110	13063.054	15700.270	16846.324	17910.297	7	54,74537	7,354634
36	3039	12735.166	15199.222	16535.055	17756.5	7	59,72333	7,550769
37	2900	13037.472	15951.611	16510.027	17584.111	5	54,695	6,89525
38	2970	14430.45	17528.025	18943.925	19798.325	9	52,20977	6,039767
39	2840	13734.108	16681.243	17297.351	18174.810	2	79,27878	10,45146
40	2849	14121.184	16928.921	17754.736	18697.026	4	78,16857	11,23881
41	2659	16732.75	19467.409	20764.886	21679.954	19	56,65563	6,679792
42	2837	15188.675	18141.125	18741.175	19813.95	8	61,68935	7,302826
43	2901	15537.5	18032.725	18387.725	19542.125	4	63,61244	7,477556
44	2838	15142.05	17586.85	18509.225	19583.225	4	79,15911	9,845556
45	2808	15928.804	18753.170	19481.414	20342.951	7	66,99622	7,938444
46	2905	16805.953	19320.674	20114.651	21335.837	3	78,17979	10,93702
47	2907	17796.891	20764.543	21735.891	22627.739	18	57,8658	5,9796
48	2683	19133.333	22116.5	22607.520	23866.708	19	59,24808	6,136154
49	3065	17350.911	20331.555	21294.6	22110.977	3	79,60959	11,30898
50	2963	17066.860	19588.534	20385.279	21107.627	4	78,35646	10,91271

Figure A.4: At least once with queue length 100

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	4348	357.4	476.86666	523.13333	564.13333	2	70,09611	9,887222
2	4044	651.125	891.9375	982.0	1086.6875	2	72,25167	10,85833
Namnruta		1052.2941	1431.7058	1613.1764	1774.7058	2	50,49842	8,284211
4	3707	1154.9411	1686.8823	1871.7058	2053.1764	2	71,92263	11,36947
5	3555	1395.7058	1920.9411	2238.9411	2434.1176	1	70,3585	12,0225
6	3621	1719.7222	2644.3888	2923.3333	3343.3333	2	74,1115	11,4475
7	3516	2051.2222	2733.3333	3230.2222	3513.0555	2	44,69273	8,015455
8	3483	2642.1052	3735.4210	4083.0	5078.5263	1	70,80227	12,83273
9	3363	2609.35	3860.7	4470.2	5047.9	1	48,31818	8,881818
10	3343	3161.05	4358.05	4958.3	5535.8	1	52,15409	9,061364
11	3615	3726.7142	5059.8095	5593.4285	6160.0476	2	71,14667	12,16292
12	3645	3949.7727	5762.4090	6052.8636	6902.1818	2	50,68583	7,49625
13	3611	4014.4090	5420.5	5916.4090	6504.4090	2	71,7632	11,5888
14	3616	4563.6086	5869.8695	6317.1739	7197.8260	3	74,466	10,7176
15	3580	4537.9565	6069.5217	6506.5217	7695.0869	3	74,4164	11,0564
16	3519	5312.0	6937.125	7566.1666	8359.5833	1	51,70615	7,116154
17	3413	5592.3478	7466.0434	7886.0869	8822.2608	1	75,72615	11,485
18	3415	6082.25	7906.2083	8767.7916	9630.5833	2	55,79185	6,638889
19	3510	6151.6	7986.36	8550.88	9234.64	3	75,71857	11,82714
20	3431	6423.2	8544.2	8900.44	9778.4	2	75,60321	11,23679
21	3386	7008.0769	8963.3846	9731.6538	11856.807	3	76,40103	11,35828
22	3236	7549.7037	9638.7037	10251.851	11334.444	2	51,54467	6,594
23	3200	8659.2758	10502.172	11789.931	13237.586	3	52,45375	6,520313
24	3273	7782.0714	10126.0	11163.035	13078.357	4	55,62156	7,539063
25	3350	8177.8214	10348.535	11175.214	12905.785	2	76,12906	11,09219
26	3253	8478.5357	10569.928	11193.678	12053.928	3	77,34677	10,81258
27	3378	8471.4827	10783.655	11468.034	13096.448	3	75,59	10,84182
28	3059	9918.1612	12302.451	13005.0	14111.870	6	51,55	6,768
29	3157	10272.812	12843.531	13578.843	15281.218	6	54,76861	7,411667
30	3158	10554.406	12980.75	13679.218	14786.718	2	49,73194	6,04
31	3041	10952.687	13376.593	14052.75	15284.062	6	50,8375	6,028333
32	3036	12113.0	14260.676	15171.911	16385.352	5	51,22658	6,262368
33	3171	10358.967	12721.387	13447.580	14557.419	3	76,71229	10,72771
34	3149	10975.875	13330.968	14143.437	14698.437	4	76,64139	10,61833
35	3109	11709.939	13977.515	14720.848	15658.0	3	76,57	10,16973
36	2858	13540.368	16714.710	17487.157	18557.552	8	56,71452	7,472381
37	2869	13812.684	16596.605	17220.684	18674.368	8	59,31833	7,132619
38	2820	13832.972	16277.432	17162.459	17874.108	3	78,23452	10,08714
39	2800	14280.526	17126.184	17513.342	18688.157	3	79,47048	10,37214
40	2730	15196.394	17389.157	18103.157	18880.657	3	77,73465	10,75023
41	2879	15432.365	18254.707	19308.0	20315.902	7	62,25022	7,632889
42	2934	14547.184	17064.026	17710.684	18170.526	6	62,79791	7,409302
43	2902	15265.75	17587.725	18416.075	19377.3	7	60,77795	7,557273
44	2775	16885.363	19990.840	20926.863	21629.931	5	62,94375	6,783542
45	2772	18476.608	21129.347	21870.869	22844.108	7	63,4122	7,0716
46	2751	17570.0	20006.431	20553.727	21694.0	18	58,98388	6,193265
47	3056	15336.125	17975.8	18618.9	19579.075	3	79,52432	10,51318
48	3025	15464.536	18317.024	19059.048	20165.292	4	78,67044	10,88156
49	2934	16623.659	19643.954	20857.0	21520.545	4	79,3425	10,84042
50	2863	19094.851	21434.127	21822.531	23253.510	11	62,85769	7,263846

Figure A.5: At least once with queue length 1000

A	B	C	D	E	F	G	H	I
queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	4318	338.86666	450.46666	501.53333	538.2	2	69,67556	10,35167
2	4067	730.625	997.25	1073.9375	1189.4375	2	53,23333	7,697778
3	3997	814.6875	1131.0	1306.8125	1419.875	2	71,26737	10,59789
4	3615	1238.7058	1797.4705	1948.2352	2224.3529	2	48,12368	7,745263
5	3643	1407.9444	2116.1666	2364.8888	2633.3333	1	47,6065	8,483
6	3567	1788.1111	2733.4444	2897.3888	3379.6666	2	49,39857	8,535714
7	3592	2072.6315	2847.8947	3322.1578	3628.4210	1	72,06952	12,21857
8	3488	2261.5263	3170.2105	3830.0526	4225.2105	1	49,74	7,81
9	3517	2644.15	4092.2	4369.35	5231.8	1	72,85818	12,675
10	3451	3021.65	4129.2	4682.1	5064.4	1	71,87455	15,09727
11	3362	3939.0909	5199.6363	5604.8181	6319.0454	3	53,37042	9,98875
12	3626	3810.2857	5160.6190	5770.7619	6384.1904	3	49,91042	7,699583
13	3594	4331.4782	5966.3913	6677.2608	7484.6521	1	49,2492	6,7548
14	3595	4364.1363	5822.1363	6344.4090	7260.5909	2	73,4124	10,9248
15	3549	4782.4347	6229.4347	6840.8260	7318.4347	2	74,2656	12,3952
16	3471	5500.6956	7031.4347	7611.1739	8447.0434	3	73,32808	11,51538
17	3361	5869.52	7666.52	8212.92	8923.6	2	53,62407	6,24037
18	3308	5793.76	7845.72	8669.24	9434.08	2	52,50778	6,424444
19	3370	7041.5	8634.1538	9479.4615	10485.538	3	51,54655	6,974483
20	3210	7776.2222	9387.2962	10326.333	10928.407	3	52,21867	6,598
21	3203	8195.9310	10856.793	11798.172	13716.103	3	49,90844	6,911875
22	3206	8142.7777	10066.703	11121.962	12214.851	3	52,97645	7,135806
23	3232	8733.8333	11339.3	12654.1	14451.1	4	52,59545	7,097576
24	3200	8466.6785	10477.928	11008.785	12702.25	4	51,61806	6,667742
25	3376	7847.5185	10203.703	10498.259	11572.148	2	76,64167	10,50433
26	3269	8566.7857	10562.357	10987.607	12068.535	2	75,96968	11,48484
27	3348	8548.9310	11079.482	11588.724	12565.413	3	75,785	10,96031
28	3231	10076.812	12378.968	13153.25	14180.5	2	53,89971	6,987143
29	3160	10267.903	12619.387	13161.935	14160.741	3	52,55629	7,451143
30	3017	11393.878	13288.939	13945.909	15431.454	5	51,18056	7,384444
31	3249	10086.25	12650.031	13828.312	15230.0	2	78,23457	10,87657
32	3196	10897.848	13412.909	13931.484	15059.848	3	77,05459	10,8873
33	3097	11607.969	13882.0	14152.030	15318.0	4	52,84297	7,476757
34	3215	11354.272	13593.151	14313.363	15942.272	3	77,1927	11,22243
35	3150	11664.628	14200.885	15455.6	16616.4	3	76,70795	11,61872
36	3112	11801.090	14220.393	14665.848	15868.030	3	78,00703	10,64757
37	3099	12589.828	14795.571	15916.057	17003.171	3	79,28462	10,39538
38	2866	16985.232	19330.209	20633.860	21196.953	12	52,94979	6,652128
39	2772	15104.175	17756.75	18754.45	19733.4	10	54,03409	7,384773
40	2894	14281.184	16826.868	17727.578	18574.578	3	79,92095	11,08238
41	2808	15234.536	18629.658	19620.024	20306.707	11	55,77244	7,075778
42	2782	15867.0	18245.675	18977.825	19772.375	2	77,68556	11,01667
43	2917	15944.380	18635.0	19679.904	20461.904	6	66,53696	8,478478
44	2857	16557.738	18983.071	19634.166	20776.047	9	64,76022	8,166957
45	2828	17517.533	20334.555	20997.511	21948.755	9	65,64857	7,93
46	2815	16719.883	19258.372	20290.302	21374.372	7	63,46362	8,254681
47	2840	16145.380	18902.642	19520.738	20430.047	4	76,86043	10,40362
48	2898	17417.954	20017.272	20965.727	21677.590	12	63,40167	7,409167
49	2873	18023.021	20854.630	22248.173	22886.326	13	61,7444	7,5744
50	2952	16853.162	19394.953	20183.395	21262.186	11	64,28362	7,657021

Figure A.6: At least once with queue length 10000

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	11156	175.4	234.2	257.4	280.93333	1	35,39824	6,8
2	11272	300.4	403.33333	448.26666	486.93333	0	31,48	6,238889
3	10014	387.375	668.125	782.8125	878.5	1	31,93556	6,363333
4	10042	521.5625	816.6875	948.4375	1096.0	1	68,32211	9,140526
5	10323	610.375	946.5	1041.625	1227.25	1	35,44778	7,267778
6	10274	739.875	1122.375	1191.9375	1380.5625	1	34,1205	6,1255
7	10592	794.75	1168.0	1313.4375	1511.625	1	72,1	9,936842
8	10276	938.47058	1493.9411	1684.1176	1942.8823	2	72,99947	9,581579
9	9102	1142.6470	1710.1764	1956.6470	2258.9411	1	31,58316	6,350526
10	9665	1164.8823	1704.1176	1913.4705	2087.8235	1	37,18632	6,518947
11	9998	1348.6470	1987.4705	2136.8823	2412.1764	1	71,3605	10,538
12	9961	1319.3529	1937.2941	2269.2352	2648.6470	1	72,502	9,8335
13	9942	1468.6666	2138.3333	2400.8333	3039.4444	1	72,8765	10,013
14	9313	1610.8333	2428.1111	2826.3333	3092.8333	1	33,865	6,0155
15	9146	1800.5555	2694.8333	3107.3333	3563.6666	1	33,4595	5,783
16	8591	2015.3333	2963.0	3413.1111	3680.0	1	36,37286	6,724286
17	8523	2114.4375	3343.625	3604.9375	4176.9375	1	35,26762	6,909524
18	8513	2325.4736	3398.0526	3742.0526	5969.2631	1	36,85	6,460455
19	8528	2326.8823	3600.4705	3801.5294	5781.8235	1	72,24591	10,58545
20	8094	2701.8333	3846.6666	4261.5555	4886.8333	1	36,73045	5,803636
21	8134	2830.3157	4038.5263	4481.8947	5274.3684	2	37,10091	6,198636
22	7858	3107.2222	4450.8333	5060.4444	5750.1111	1	37,91	7,197727
23	7658	3321.7222	4808.0	5122.9444	5819.8333	1	39,93727	6,908182
24	7853	3143.2105	5127.6842	5958.7368	9921.5263	1	42,73304	6,225217
25	7789	3255.7222	4983.1666	5173.2222	5932.6666	2	41,4587	7,093913
26	8176	3236.3684	4812.3684	4987.2105	5819.4210	2	39,41957	6,381739
27	7883	3464.4444	5288.6111	5613.0	6587.8888	1	42,91739	6,956087
28	7387	4305.45	5972.4	6772.6	8828.5	1	40,1584	6,7368
29	7749	4178.0	6166.8421	7339.5263	10626.789	2	52,6968	8,4228
30	7725	3989.6111	5971.0555	6274.9444	8490.8333	1	72,82833	11,60667
31	7310	4439.5789	6849.4736	8211.2105	11021.210	2	42,9348	6,5916
32	7369	4617.1578	6832.3684	7168.8947	9240.0	1	42,1452	6,7524
33	7611	4430.1666	6229.0	7245.5	10779.611	2	73,278	11,4576
34	7165	4889.4736	7248.6315	7660.3157	9822.4736	1	43,39115	6,193077
35	7889	4408.05	6337.15	6772.25	7704.2	2	71,5512	11,4412
36	7160	4975.6470	7825.7647	8642.6470	10813.941	2	47,56	6,835
37	6917	5387.1875	7363.6875	8607.875	9512.75	3	47,9688	6,856
38	6711	5511.7894	8105.5789	8484.8421	9571.2105	1	45,02385	6,689615
39	6795	5432.3888	8282.9444	8877.7777	9798.3333	2	49,90769	6,987692
40	6413	5996.0	8588.1111	10345.555	12681.222	1	48,41593	6,814444
41	6658	5842.4444	8253.5	9189.8888	10077.111	2	74,18269	11,03231
42	6292	6796.2222	9573.9444	10292.055	11901.777	3	52,31037	7,493704
43	6027	7331.3333	10555.888	11102.888	12343.777	2	50,29429	7,397143
44	6799	6440.5294	8696.4705	9956.4705	11678.588	2	55,46556	7,405185
45	6570	6928.1666	9617.0	11163.777	13189.277	3	51,67143	7,740357
46	6652	6960.7777	9989.0	11899.5	13708.388	3	54,10828	7,415517
47	6917	7090.3157	10537.315	12017.0	14055.578	2	51,81379	6,580345
48	5966	8014.7777	10864.5	12135.5	13642.055	4	46,681	6,002333
49	5783	8304.1176	10757.705	12362.882	13057.764	3	48,4431	6,609655
50	6572	7379.6666	10869.722	11676.333	13530.666	3	51,83138	7,416552

Figure A.7: At most once with queue length 10

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	11728	149.73333	206.33333	227.53333	249.46666	1	75,92824	7,394118
2	11428	261.73333	374.8	417.93333	477.6	1	75,34176	8,842353
3	10765	608.875	798.75	881.4375	935.1875	1	73,44056	9,140556
4	10250	615.5	852.3125	917.8125	1038.125	1	37,65833	6,252222
5	10448	658.1875	931.0	984.4375	1109.0	2	36,99667	5,649444
6	10341	844.8125	1208.375	1347.1875	1533.0625	1	36,51722	6,437778
7	10345	882.94117	1313.8235	1536.7058	2092.0	1	37,29316	7,286316
8	9844	1022.8823	1521.3529	1647.1176	2030.6470	2	33,81158	5,89
9	9540	1199.4705	1779.2352	1867.6470	2185.8823	2	32,61579	6,234211
10	9642	1182.5294	1705.0588	1997.3529	2211.7058	1	31,70632	5,477895
11	9468	1419.8235	2051.8823	2256.4117	2567.6470	1	34,868	5,588
12	8851	1434.7058	2286.3529	2559.5294	2871.5882	1	31,4325	8,4635
13	9463	1451.8333	2208.0	2317.9444	2715.5	1	33,5945	5,9455
14	9215	1793.8333	2453.1111	2844.5	3133.9444	1	31,5095	5,802
15	9588	1748.3333	2555.6666	2967.7777	3847.7222	1	32,3345	5,472
16	9527	1798.1052	2690.4210	3081.5789	5978.4736	2	73,4019	10,8819
17	9228	2161.1052	2994.4210	3497.7894	4588.6315	1	31,22318	5,606818
18	8519	2248.3888	3339.2222	3550.9444	4127.6111	2	37,69238	6,544286
19	8137	2494.1764	3877.8235	4023.7058	5486.8235	2	34,22091	5,567273
20	8294	2538.5789	3753.3157	4015.9473	4854.2631	2	35,62318	5,984091
21	8122	2671.8421	4055.0	4480.6842	8536.5263	2	37,03348	6,056087
22	8342	2630.0	3921.8	4029.05	4720.65	1	38,16273	6,359091
23	9072	2630.7368	3857.5789	4140.6315	5096.9473	1	74,85455	10,325
24	7961	3147.8947	4801.1578	5423.2105	7962.7894	2	42,09	7,677826
25	8322	3054.2105	4508.1052	4790.4736	5300.2105	1	74,63455	10,64727
26	8129	3318.9473	4973.2631	5328.0526	7918.7368	1	43,21783	7,679565
27	8043	3553.5	5180.0	5389.4444	6311.3333	2	73,0187	11,78783
28	8114	3746.7222	5432.5555	6170.6666	7240.3333	2	41,86417	6,660417
29	7647	4026.9444	5968.9444	6523.4444	7228.7777	3	42,83625	7,212917
30	7955	3774.6666	5851.7777	6070.8888	6950.6666	2	42,22208	7,036667
31	7883	3963.2222	5717.0555	6350.5	8398.7777	2	74,89917	11,19958
32	7759	4058.7777	6432.2777	6683.1666	7560.0555	2	74,00833	12,31292
33	7808	4405.3888	6021.0555	6738.4444	7292.0	2	75,08292	10,52417
34	7401	4572.6111	6886.7777	7244.1111	8580.2222	2	43,716	6,6264
35	6808	4987.9523	7059.8571	7760.3809	8862.7142	3	75,61	10,48393
36	7621	4981.9473	6783.4736	7591.0	9287.4736	2	76,84	9,2636
37	7256	5338.0	8097.8421	8471.7894	9617.6315	3	46,43577	6,785
38	6947	5325.0	7676.1666	8716.8888	9904.9444	3	45,2	7,761154
39	6910	5743.0	8386.5789	8818.4210	9984.7368	2	47,95577	6,962308
40	7203	5695.4	7541.15	8260.85	9085.5	2	52,31308	7,675
41	7351	5211.0526	7700.6842	8309.6842	10388.368	2	75,98	11,21731
42	6925	5635.2105	7913.0526	8748.3157	9606.4210	3	72	10,57185
43	7199	6040.9473	8126.4210	8971.0526	10140.947	3	74,11852	10,90889
44	6848	6134.1578	8499.4210	9424.5789	11089.894	2	75,55111	10,84852
45	6896	6698.5263	9245.9473	10551.631	12853.631	2	50,065	7,570714
46	6946	6551.2105	8764.8947	9528.4210	10545.789	2	74,17889	10,73111
47	7050	6433.1666	9313.6111	9811.8333	10744.666	2	73,83893	10,765
48	6971	6216.7777	8961.5	9589.8888	10800.722	3	73,17071	10,95893
49	6998	7492.8636	11174.772	12839.954	14143.5	3	51,39226	8,512903
50	6817	6927.8	9723.5	10651.05	12483.15	2	49,03931	6,035172

Figure A.8: At most once with queue length 100

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	11008	105.46666	169.0	193.73333	219.86666	1	33,65	6,060882
2	10602	288.4375	408.625	453.9375	501.125	1	31,79222	6,294444
3	10762	472.25	636.5	695.9375	763.625	1	73,06389	8,331111
4	10497	659.375	891.5625	994.625	1107.9375	1	72,17944	9,748333
5	10750	867.8125	1201.75	1335.9375	1503.875	1	34,68526	7,868421
6	10562	708.375	1102.75	1277.9375	1457.5625	1	35,245	7,431667
7	10339	853.88235	1367.2352	1564.4705	1753.8235	1	35,04316	7,595263
8	10604	883.35294	1406.7058	1509.1764	1741.2941	2	36,22	7,053684
9	10333	1083.1764	1491.9411	1706.5882	1834.1764	1	72,29579	10,00842
10	10027	1263.8235	1900.4117	2154.1764	2582.4117	2	35,89842	6,192105
11	9525	1356.5882	1894.0	2185.5294	2437.4117	3	32,4255	7,538
12	9756	1569.8823	2329.2352	2437.6470	2902.0	4	33,3475	5,7905
13	9618	1368.9444	2173.3888	2424.5555	2729.5555	2	32,809	5,9865
14	9462	1612.9444	2391.4444	2558.2222	2983.4444	1	71,04381	10,83667
15	9467	1707.5	2490.0555	2587.1666	3004.2222	2	70,56048	10,41857
16	8938	1820.2222	2815.0	3115.5	3432.8888	2	33,04857	6,489524
17	8356	2492.9444	3348.9444	3588.6666	4010.1666	5	35,33333	6,079524
18	8737	2199.8888	3433.1666	3671.0	4182.1111	2	34,62333	5,861905
19	8113	2733.6470	3924.1176	4513.2941	4995.6470	5	35,83714	5,588571
20	8540	2531.9444	3924.4444	4399.6666	6738.8333	6	40,00591	6,729091
21	8565	2435.7222	3785.6666	3961.0	4633.1666	5	40,86952	7,488571
22	8257	2790.0	4194.6842	4366.5789	6324.6842	6	72,25652	11,13696
23	8555	2760.0526	4230.5789	4841.5789	6338.7894	6	42,36682	6,848636
24	8254	3142.4210	4511.8421	5276.2105	8130.7894	8	41,97304	6,646522
25	8310	3078.0555	4762.0	5212.6111	6001.3333	8	45,57913	7,255217
26	8383	3043.2105	4724.4736	5117.6315	6328.9473	5	74,56435	9,694348
27	8136	3469.5	5075.0555	5370.7777	6128.4444	7	40,54522	6,515652
28	8151	3461.6842	5104.3157	5668.1578	8264.2105	7	75,81522	10,54565
29	8175	3546.6315	5540.7368	6325.9473	7418.7368	7	74,42583	10,99458
30	8130	3865.6315	5702.7368	6289.3684	7988.7368	7	76,1925	11,93667
31	7881	3948.2105	5948.2105	6416.8947	7316.6315	8	75,64958	11,35625
32	7893	4080.3684	6448.8421	7476.6842	11090.052	9	44,6752	7,162
33	7607	4452.8947	6750.9473	7198.5263	9158.6315	9	43,0596	6,3068
34	7415	4569.6111	6831.0555	7159.2777	8237.5	11	43,5208	7,5472
35	7286	4882.1052	7295.2631	7592.2105	9239.7894	11	48,33615	6,349231
36	7034	5235.2631	7588.0526	7950.4736	9002.3157	12	42,05962	6,972308
37	7965	4728.7894	6874.4736	7526.5263	8860.7894	9	49,6588	7,8748
38	7351	5348.1052	7828.3684	8443.9473	10796.526	11	47,47	7,518846
39	6939	6008.5263	8603.3157	8994.6315	10768.263	10	48,31963	7,44037
40	7397	5230.1052	7436.2105	8268.9473	9101.0526	9	75,965	10,74423
41	7508	5746.15	8019.6	9218.25	10968.55	12	51,82704	8,536667
42	6948	5710.9	8367.9	9051.6	11249.9	12	50,63259	7,123333
43	6910	5880.4210	8398.7368	9318.0	11487.0	12	52,19185	8,213704
44	7134	6010.9473	8173.7368	9294.8947	10449.947	10	75,09074	9,946296
45	6107	7820.7222	10848.5	11643.111	12829.722	13	46,97148	7,58
46	6301	7116.2631	9426.5263	10693.421	13060.210	11	77,19964	10,98357
47	7276	6409.65	8988.95	10052.3	12423.9	11	76,46357	11,04143
48	7031	6279.7368	8713.2105	10146.210	11649.263	12	76,22107	10,25964
49	6389	8028.4285	10723.047	11737.571	12790.619	13	46,58367	7,371667
50	6577	7268.4736	9817.0526	11973.736	13543.526	12	75,11931	10,69345

Figure A.9: At most once with queue length 1000

queue	msg/s	min	avg	95th	99th	disk MB	cpu%usr	cpu%sys
1	11151	185.4	253.66666	280.33333	303.26666	2	32,91647	6,192941
2	10950	305.33333	448.66666	500.4	559.0	2	27,62611	6,834444
3	10813	431.625	575.1875	639.5	701.25	2	34,31389	7,585
4	10938	560.375	862.0625	985.625	1113.4375	2	74,12389	10,11056
5	11093	772.8125	1055.25	1177.0625	1331.1875	2	72,39158	9,104737
6	10476	839.625	1199.875	1330.5	1586.6875	2	34,07842	7,247895
7	10532	840.8125	1264.0625	1356.25	1570.4375	2	35,72	6,397895
8	10442	1157.1176	1702.1176	1882.5294	2124.2941	2	33,98947	6,600526
9	10062	1105.4117	1654.5882	1764.9411	2034.5882	2	32,43579	6,073158
10	10122	1117.4705	1647.2352	1866.1176	2032.3529	3	73,62632	10,25474
11	9669	1240.2941	1892.4705	2213.2352	2496.1764	5	33,195	6,8785
12	9545	1401.6470	2004.4117	2118.0588	2450.1176	5	32,664	7,0465
13	9208	1641.3888	2261.2777	2569.6111	2785.1111	6	31,8205	5,881
14	9209	1684.6666	2409.4444	2842.6666	3449.8333	6	32,8025	6,9485
15	8608	2038.4210	2843.1578	3370.0526	6296.0526	7	32,01	5,209524
16	8933	1884.7777	2832.0	3117.6111	5902.8333	6	74,30286	10,28857
17	9089	2028.5882	3009.4117	3155.7058	4068.8235	7	73,26857	10,32143
18	8444	2283.5263	3341.2105	3575.3684	4953.9473	7	35,4981	5,424762
19	8352	2509.6666	3700.3333	4194.7222	4677.5	7	34,64136	6,395455
20	8719	2601.7894	3773.6315	4138.5263	5211.4736	2	32,76364	6,289545
21	8187	2483.8947	3963.1578	4169.3157	4846.2631	7	36,92045	6,485909
22	8078	2812.85	4434.45	4771.55	9523.8	8	37,02	6,016957
23	8037	3176.95	4648.05	5659.35	8871.25	9	38,465	6,72375
24	8236	2989.1578	4558.3157	4732.6842	6149.7368	9	39,41565	6,887391
25	7780	3139.45	4633.85	4998.25	5824.25	10	39,00391	6,541739
26	8497	3000.0526	4636.4210	4807.4210	5775.1578	14	73,26174	11,1513
27	7093	4168.9565	5904.5652	7261.6521	9797.5217	10	36,08185	5,477037
28	7933	3986.3157	5567.6315	6331.7894	6995.5263	12	50,12083	7,751667
29	8140	3438.8333	5356.1666	5588.5	6496.0	11	75,03696	10,02565
30	8152	3712.2631	5547.8947	5838.7894	7681.3157	11	72,6412	10,2104
31	7691	4316.95	6024.75	6425.95	8837.95	14	4,785846	0,729596
32	7166	4409.35	6494.35	6843.7	7715.15	11	40,946	6,2992
33	7429	4702.6190	6795.7142	7443.6190	10244.095	12	44,88846	6,927308
34	7619	4706.6666	7195.3809	7743.8571	10781.095	13	45,57423	6,541538
35	7524	4851.0	7110.5714	7727.4285	9852.8571	12	45,48538	7,121154
36	7459	5025.95	7381.05	7911.85	9650.45	12	45,65385	7,681923
37	7071	4998.5	7317.3	8103.85	9266.85	13	44,47654	7,345
38	7406	4983.65	7456.7	7942.35	9536.15	13	49,43577	7,486154
39	7290	5116.7272	7281.0	7890.1818	9455.1363	13	49,46154	8,070769
40	7201	5570.2105	7661.6315	8529.3157	9154.1578	14	74,51077	11,20577
41	7302	5818.2272	8441.2727	8925.7272	11840.5	14	50,81536	6,78
42	6898	6239.3636	8878.3181	9680.4090	12609.045	16	52,54241	7,266207
43	7102	5859.3636	8079.0454	8764.3181	9742.9090	15	51,4837	7,945556
44	7158	5922.6190	8034.5238	8912.2380	10190.666	15	51,74222	8,344444
45	7111	6105.1363	8638.9090	10171.227	12611.409	16	50,4631	7,335517
46	6753	6555.9523	9190.3809	9861.9523	11039.571	17	49,06071	7,476071
47	6910	6384.15	8637.45	9852.45	10532.05	17	74,65607	10,43679
48	7009	6891.4761	9002.0952	9947.4761	10878.047	16	75,61321	10,49429
49	6984	6879.4285	9152.6666	10535.095	12452.142	16	73,65414	11,05655
50	6701	7985.0869	10847.869	11526.086	12746.782	17	47,89367	7,273667

Figure A.10: At least once with queue length 10000