Université Catholique de Louvain
Ecole Polytechnique de Louvain
Computer Engineering Departament

RoQ: Message Log Management with Zero-copy and Memory Log

Supervisor:
Peter Van Roy
Co-Supervisor:
Sabri Skhiri
Reader:
Nam-Luc Tran

Master's thesis submitted for the graduation of
Master in Computer Science (120 credits)
option Software Engineering and Programming Systems
by Cristian Voicu

Louvain-la-Neuve
January 6, 2014

Contents

Acknowledgments

I would like to thank Pr. Peter Van Roy for the support and trust accorded throughout the writing of this thesis.

I would also like to thank Sabri Skhiri for his guidance and availability during the whole thesis.

I thank Nam-Luc Tran for reading this thesis.

My gratitude also goes to my parents who sustained and encouraged me during my studies.

I would like to thank Teodora and Jean for the support accorded during my studies.

Finally, I would like to thank my sister Andra for the aid accorded during the writing of this thesis.

Abstract

Handling messages at runtime is good, but having the possibility to handle previous messages is better. The RoQ elastic message queue handle messages at runtime, but if a subscriber wants to retrieves messages it is not able to do it. This thesis aims to bring the RoQ system persistent by logging the messages. The system architecture is designed to be reliable in terms of message storage. The implementation proposed in this thesis is based on zero-copy and memory log techniques. Considerings the message persistence, performances of the RoQ with Message Logging System are not significantly reduced compared to RoQ without MLS.

Chapter 1

Introduction

The goal of this chapter is to identify and to set up the research problem addressed in this thesis. For this purpose, this chapter is structured into three sections: the first section introduces the context of the problem addressed in this thesis, the second section defines and motivates the central goal to be achieved by this thesis, and the third section outlines the thesis organization.

1.1 Problem Context

This thesis is interested in the area of distributed systems, specifically by distributed storage systems, a domain that today receives more attention for several reasons: on one hand, the price of technologies required to run a distributed system has been significantly reduced over the past few years, on the other hand, more efficient technologies are progressively appearing to support the deployment of distributed storage systems. A definition of distributed system is given by Tanenbaum & Van Steen in their book *Distributed Systems: Principles and Paradigms* [1]:

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

In the context of this thesis, this definition is complemented with the CAP theorem proposed by Eric Brewer [2]:

> *The theorem states that a distributed system can't handle simultaneously all three guarantees: consistency, availability, partition tolerance.*

This theorem starts as a conjuncture in [2] and become a theorem with the formal proof of the conjuncture made by Seth Gilbert and Nancy Lynch in [3].

For this purpose, in the overall domain of distributed systems, this thesis is focusing on the distributed storage of messages which pass through RoQ.

RoQ (pronounce rɒk'juː, as in "Rock You") *is an elastic message queue that enables to support a significant number of publishers and subscribers. In RoQ a logical queue is composed by a set of independent exchanges. If an exchange is overloaded, RoQ allows to spawn new exchanges and relocates a subset of the producers. This is completely transparent for the subscribers and publishers. RoQ is an implementation of Elastic Queue Service.*[4]

Elastic Queue Service (EQS) *is a message bus designed to enable elastic scalability, performance and high-availability. Its design is targeted for a deployment on a cloud computing infrastructure using commodity hardware. EQS contributes on the architecture level of the Message Oriented Middleware.* [5]

Message Oriented Middleware (MOM) *is a layer which allows software components that have been developed independently and that run on different networked platforms to interact with one another. It is when this interaction is possible that the network can become the computer.* [6]

Defining the problem

RoQ is not persistent. Indeed, the messages that pass through RoQ are immediately sent to the subscribers. If a subscriber wants to receive again any of previously received message, RoQ cannot satisfy his request since no message are preserved.

Solving RoQ persistence is

- An important problem: when message persistence is ensured, messages are written to logs. If a subscriber should be restarted after any problem (e.g., a software/hardware failure, a network failure between the components of the system), it recovers these messages from the logged data. When message persistence is not ensured, messages can not be retrieved by subscribers.
- A complex problem: a log manager could be responsible for managing a potentially huge amount of messages between a large amount of components. The message storage should be reliable and should keep the elasticity guarantee of RoQ. Beside this, ensure a high throughput and a low latency.
- An open problem: different tools exist for ensuring message persistence for different systems, but there are no tools which ensure all previous guarantees for message persistence in RoQ.

1.2 Thesis Statement

In order to address the aforementioned problem we define the thesis statement as follows:

> *The goal of this thesis is to specify, design, implement and test a message logging system for RoQ that enables reliable distributed message storage while allowing subscribers to retrieve any message on demand and that does not affect the RoQ performance.*

Message Logging System allows saving messages into a file or directs the messages to other devices.[7]

Reliable distributed message storage Is the ability of a distributed system to store messages which persists in cases of failures.

Message retrievability Is the ability for a subscriber to request and to receive previous messages from the broker.

In the development life cycle of the message logging system, we will focus on the specification, the design, the implementation and the testing of the message logging system.

1.3 Thesis Outline

The description of all tools, technologies, libraries used in the design of the architecture and in the implementation is realized in the Chapter 2 - State of art. In the same chapter, two existing message logging systems, Kafka and Zper, are described.

The design of the architecture is realized in Chapter 3 - Message Logging System. The architecture proposed in this chapter is expected to be generic - it can be implemented in different programming languages. However, some technologies are part of the design and should be taken in consideration in the implementation phase.

Chapter 4 - A framework is the result of the implementation of the architecture of MLS which is described in the third chapter. The integration of Message Logging System framework in RoQ is done taking into account the fact that the RoQ will evolve.

Chapter 5 contains several tests of both the original system and the integrated system considering a relevant range of values for some varying important parameters. Tests are realized in order to observe the reliability of the MLS while failures are introduced. The results prove that the introduction of MLS does not affect the overall performance of RoQ system.

A summary of the contributions brought by this thesis is provided in Chapter 6, followed by a critical analysis of benefits vs. shortcomings of the implemented MLS that leads to some avenue for this work.

Chapter 2

State of art

The goal of this chapter is to introduce some techniques considered useful for addressing the problem of MLS implementation.

The chapter contains 7 sections. In the section 2.1 and 2.2 two techniques of memory performance optimization are described. The section 2.3 introduce the ZMQ library and enunciate its functionalities. In section 2.4 the RoQ message queue is explained. In section 2.5 three storage systems based on key/value are presented: Hbase, Cassandra, Infinispan. In section 2.6 and 2.7, Kafka and Zper, two existing log systems are separately analyzed while in the last section 2.8 a comparison between Kafka and Zper is discussed.

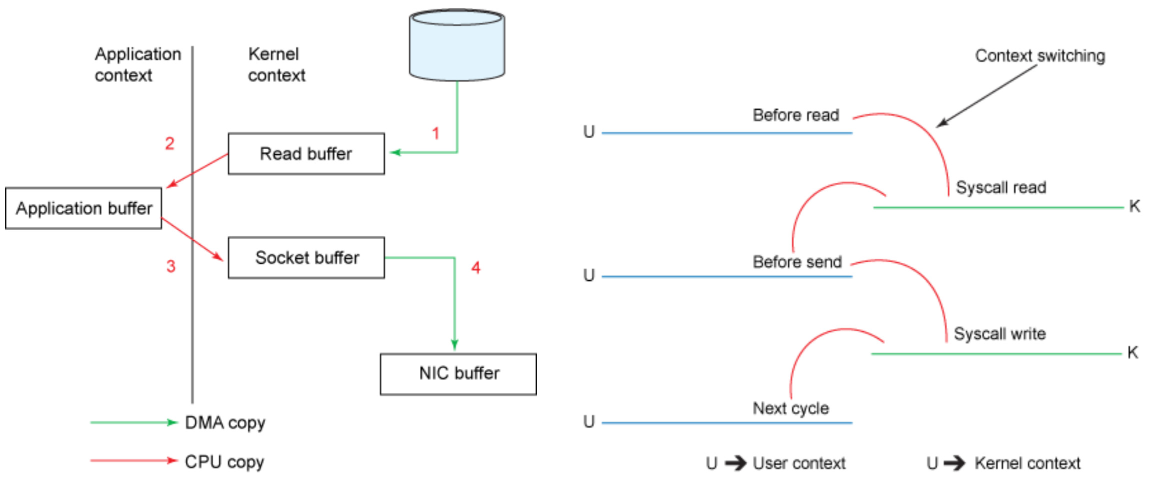
2.1 Zero-copy

Definition: *Zero-copy avoid redundant data copies between intermediate buffers and reduces the number of context switches between user space and kernel space.* [8]

Copying data from a place to another in memory is costly. We know also that changing the context is costly [9].

Consider the following situation: a file stored on disk is sent to another computer through the network.

Before explaining how zero-copy works, the traditional data copy approach is described. In Figure 2.1a the first operation `(1)` copies the data from the file (on the disk) to the read buffer of the kernel space memory. Then `(2)`, the read buffer is copied to the application buffer of the application space memory. In this way the application can manipulate (verify, change, etc...) data before sending. Afterwards `(3)`, to send the data, the application buffer is copied again to the kernel space memory in the socket buffer. Finally `(4)`, the socket buffer is copied to the buffer of the network card. Each time when a system call appears, the context must be changed from the user context to the kernel context. Notice that in the traditional approach there are four data copies and four context changes. [8]

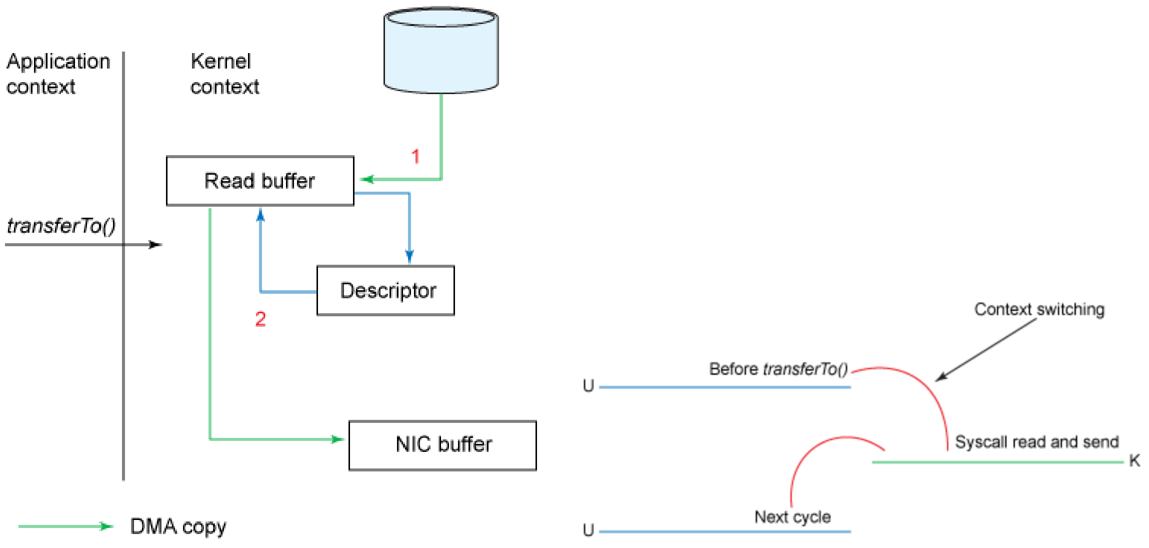


(a) (b)

Figure 2.1: Traditional data copy and context switch. [8]

Next, the zero-copy technique is described. This technique uses only the kernel space to transfer the data of a file through the network. In the Figure 2.2a the first operation is the same as in the traditional data copy technique. Then, the read buffer is copied directly to the network card buffer without passing through the application context. This is possible because the read and send call systems are done one after another without changing the context. [8]

There are only two data copy operations and two context change with the zero-copy technique compared to four data copy operations and four context changes in the traditional data copy technique, which represents an interesting difference.



(a) (b)

Figure 2.2: Zero-copy data transfer and context switch. [8]

2.2 Memory log

In this section, firstly, the technique memory log is defined, then advantages and inconveniences of using this technique are enunciated.

> *The memory log allows a program to maintain a rotating log of recent messages in memory.* [10]

Saving messages in memory may be done in a very fast way on the latest generations of memory [11]. The memory log technique stands in collecting messages into a buffer. The main advantage for a system which use such technique stands in the fact that there are no looses of performances because there are no I/O syscalls.

The main inconvenient is the memory size limit and that means there are a limited number of messages that can be saved in memory. Related to the hierarchy of memory [12], the size of the memory (RAM) is smaller then the size of the disk. Therefore, the number of messages that can be stored in memory is smaller than the number of messages that can be saved on disk. Keeping a small number of messages is the big inconvenient of the memory log technique.

2.3 ZMQ

In this section, firstly, the library ZMQ is described. Next, the deployment of this library to some applications is presented, followed by an example.

2.3.1 A tool for distributed applications

ZeroMQ is an opensource project, started at iMatrix. The goal of ZMQ is to provides tools for deploying distributed systems in a easy, fast and efficient way.

Definition: *ZMQ (also seen as ZeroMQ, 0MQ, ØMQ) looks like an embeddable networking library but acts like a concurrency framework.*[13]

This framework supports many kinds of connection patterns [14]:

- Request-Replay : It is the classical client/server model, for each request of the client there should be a response of the server.
- Publish-Subscribe : In this pattern the server pushes the messages to the clients that listen to the server. The messages are sent asynchronously.
- Parallel Pipeline : A “ventilator” produces tasks (messages) and send them to “workers” that process those tasks. Finally a “sink” collects results from workers.
- Fair Queuing : When many producers send messages to a consumer, the fair queuing pattern ensures that each producer sends messages in a fair way.
- Exclusive Pair: This connects two threads in a process.

2.3.2 Deployment in applications

ZMQ is a framework designed for distributed applications. When designing a distributed application we pay attention on the functionality and on the service given by the application. Therefore, we do not want to be concerned of how messages are transmitted in the transport[1] layer. The importance of ZMQ is that it handles problems that can occurs in the transport layer and provide a reusable messaging system.

The steps for using a socket are:

- create and initialize the socket with one of the socket type : pub/sub, req/rep, dealer, downstream, forwarder, push, pull, queue, router, streamer, upstream, xpub, xsub, xreq, xrep.
- bind/connect : choose the transport type (INPROC - in process communication model, IPC - inter-process communication model, MULTICAST or TCP)
- send/receive : using different methods for sending/receiving bytes array, Strings, ByteBuffers or zero-copy technique.

An essential advantage is that ZMQ supports zero-copy technique.

According to the evaluation proposed by Dworak et all. in *Middleware trends and market leaders 2011* [15] the ZMQ has the highest score among others technologies which has been subject to consideration and evaluation: COBRA, Ice, Thrift, YAMI4, RTI, Qpid.

2.3.3 Example

The next example illustrates the Publish-Subscribe pattern of ZMQ sockets.

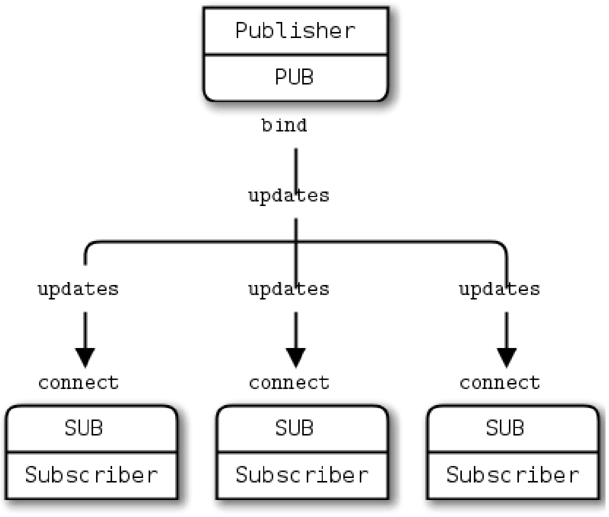


Figure 2.3: Publish-Subscribe pattern.[14]

[1]OSI model

The publisher creates and initiates a ZMQ socket of type PUB then waits for connections. The subscribers create and initiate a ZMQ socket of type sub. Afterwards, the subscribers connect to the publisher and they subscribe (register) for one or many “topics” that they will follow. In this way the messages are filtered at the level of ZMQ. Then, they start waiting for messages in a loop. The publisher start sending (updates) messages to subscribers. The subscribers receive messages of topics that they choose to follow.

Notice that the messages are send asynchronously between the publisher and subscribers. However, to ensure that the subscribers are connected before the publisher start sending messages, a synchronization technique is included. In case of using TCP and a subscriber is slower that others, there is a technique called “high-water marker” allows that publisher is not slow down. [14].

2.4 RoQ: Elastic message queue

In this section, the novelties in the message queue field and the role of RoQ are enunciated firstly. Then, the RoQ’s components are described.

2.4.1 The role of RoQ

The RoQ project started at Eura Nova[2]. Now it is turned into an open source project.

RoQ is the implementation of the Elastic Queue Service. The Elastic Queue Service is a layer of Message-Oriented Middleware, as mentioned in the introduction, section 1.1.

The novelty, that RoQ comes with, is his design to support elastic scalability: [16] *Traditionally, MOMs are not designed to support elastic scaling. This means that in a cloud context, they may very quickly become a bottleneck in terms of performance. RoQ has been designed from day 1 to answer this problem. Its architecture is elastically scalable. This includes three properties:*

- *When required, the capacity of the system will be increased automatically.*
- *This capacity increase has no impact on the global performance.*
- *When the load decreases, the system will scale down to avoid using unnecessary resources.*

Next, the description of RoQ given in the introduction, section 1.1 ,is recalled here:

RoQ is an elastic message queue that enables to support a significant number of publishers and subscribers. In RoQ a logical queue is composed by a set of independent exchanges. If an exchange is overloaded, RoQ allows to spawn new exchanges and relocates a subset of the producers. This is completely transparent for the subscribers and publishers.[4]

[2]http://euranova.eu

2.4.2 The RoQ's components

In this section each component of RoQ is explained.

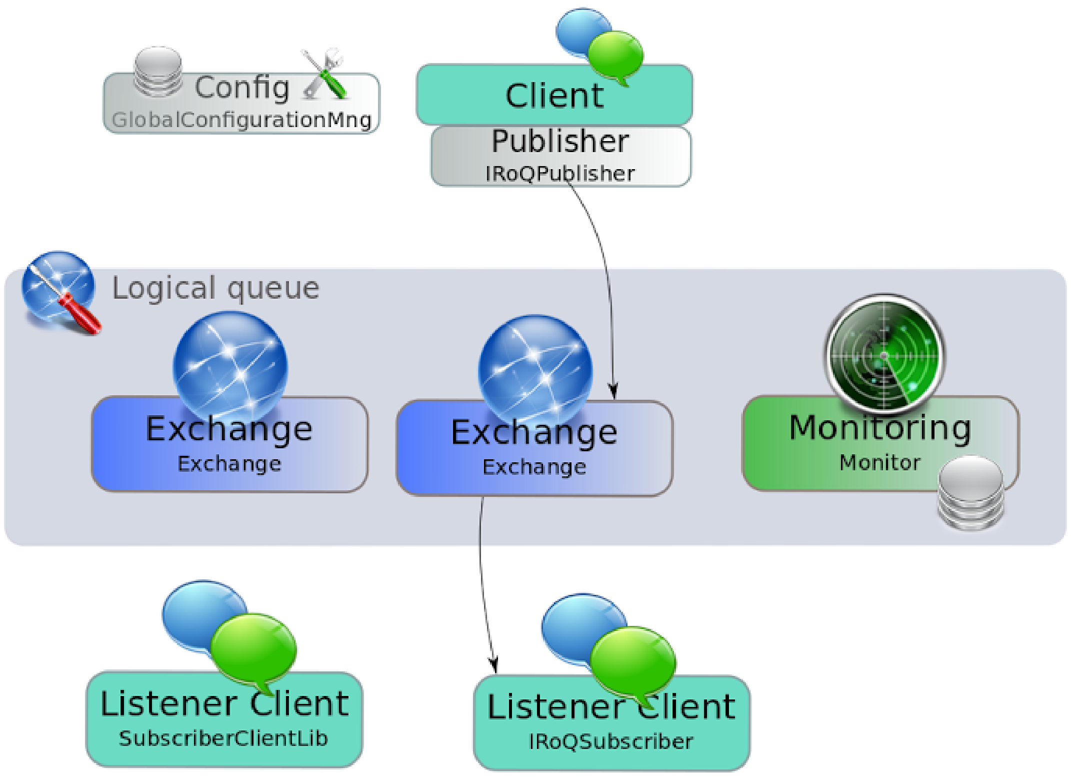


Figure 2.4: High level view of RoQ.[17]

The implementation of RoQ is developed in Java on top of ZMQ framework.

The Global Configuration Manager (GCM) is responsible to handle the global configuration of the system.

The Logical Queue is composed by a set of exchanges and a monitor.

The Exchange intermediates the changes of messages between publishers and subscribers.

The Monitor *tracks the state of the exchanges through a set of KPI such as the message throughput per second. As soon as the KPI threshold is reached, the monitor starts the re-allocation process to spawn a new exchange and to re-locate the most active publishers.* [17]

The Publisher is the producer of the message.

The Subscriber is the listener of a producer and receives the messages of the producer through the Exchange.

2.5 Distributed hash table (DHT)

The intent of a DHT is to interconnect autonomous and homogeneous nodes in order to result one coherent system [18]. The DHT easily handles the joins and leaves of nodes, allowing to scale to a high number of nodes. The information is stored in key value pairs and spread through nodes. A node has access to all stored data through the key.

Three systems based on DHT are presented next: HBase, Cassandra, Infinispan.

2.5.1 Hbase

Apache Hbase is a distributed, scalable, big data store. It is based on Google Bigtable and is build on top of Hadoop File System. Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers.[19]

Hbase guarantees ACID transactions and compliance (within a single row). Acid compliance is when acid is guaranteed by using locks (multi version concurrency control and time-stamping)

ACID is defined in [20] as:

- *Atomicity*: transactions complete or none complete.
- *Consistency*: only valid data is written. Any row returned by the scan will be a consistent view (i.e. that version of the complete row existed at some point in time)
- *Isolation*: parallel transaction do not impact each other's execution.
- *Durability*: once transaction committed, it remains.

Considering the CAP theorem, Hbase is CP: strongly consistency and partition tolerance.

Hbase operation

In this subsection the role of each component in Hbase is described as seen in [21].

Data is stored in Region Servers. As Hbase uses the key value concept, a key is needed to locate the proper server. This mapping is stored as a table of each server.

The protocol when a client do operation on database is:

- Locate the corresponding region server, the client contacts in this order:
  - Master Server who replies the Region Server that have the Root Region table.

  - Region Server who replies the Region Server that holds the Meta region table who contains a mapping from user table to region server.
  - Second Region Server (Meta region table) to find the Region Sever that handle the User Region (which contains the mapping from key range to region server).
  - Third Region Server and ask for another Region Server that handle the key that it wants to lookup.
  - Region Server that hold data at key that it wants.
- The client caches the result of the four steps above, so it doesn't need to go through those steps each time.

Hbase save data on disk like RDBMS in column-oriented approach. The difference between Hbase and the traditional consist in key-based access to a specific cell data provided by Hbase. [21]

2.5.2 Cassandra

Cassandra is distributed scalable database which handle large amount of data [22]. In the report Solving Big Data Challenges for Enterprise Application Performance Management [23] the next affirmation is made: *In terms of scalability, there is a clear winner throughout our experiments. Cassandra achieves the highest throughput for the maximum number of nodes in all experiments.*

Cassandra is under the license Apache 2.0 and developed by Apache software foundation and DataStax [24]. The language for working with Cassandra is CQL. However, there are clients which implements APIs in Java like Hector and Thrift which allow to connect to Cassandra.

The main difference of using Cassandra instead of Hbase is the decentralization. Every node in the cluster has the same role. There are no centrals points of failure. Another difference is in terms of consistency: Cassandra is eventually consistency [24], while HBase is strong consistency [21].

The short description made by DataStax cover the mains features [24]:

- *Elastic scalability – Allows you to easily add capacity online to accommodate more customers and more data whenever you need.*
- *Always on architecture – Contains no single point of failure (as with traditional master/slave RDBMS's and other NoSQL solutions) resulting in continuous availability for business-critical applications that can't afford to go down, ever.*
- *Fast linear-scale performance – Enables sub-second response times with linear scalability (double your throughput with two nodes, quadruple it with four, and so on) to deliver response time speeds your customers have come to expect.*
- *Flexible data storage – Easily accommodates the full range of data formats – structured, semi-structured and unstructured – that run through today's modern applications. Also dynamically accommodates changes to your data structures as your data needs evolve.*

- *Easy data distribution – Gives you maximum flexibility to distribute data where you need by replicating data across multiple datacenters, the cloud and even mixed cloud/on-premise environments. Read and write to any node with all changes being automatically synchronized across a cluster.*
- *Operational simplicity – with all nodes in a cluster being the same, there is no complex configuration to manage so administration duties are greatly simplified.*
- *Transaction support – Delivers the "AID" in ACID compliance through its use of a commit log to capture all writes and built-in redundancies that ensure data durability in the event of hardware failures, as well as transaction isolation, atomicity, with consistency being tunable.*

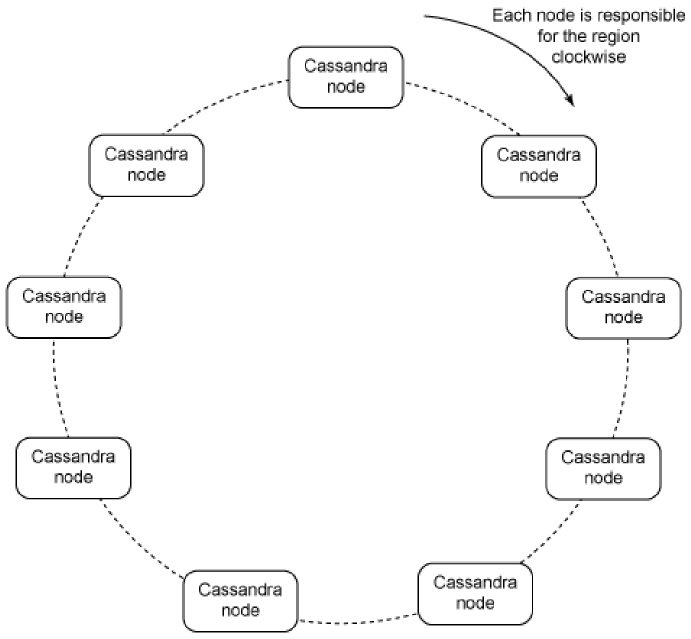


Figure 2.5: Cassandra cluster.[25]

In what follow, the two partitioning methods handled by Cassandra are presented [26]:

TOKENS: *A token is a partitioner dependent element on the ring. Each node has a unique token. Each node claims a range of the ring from its token to the token of previous node on the ring.*

PARTITIONING: *Map from Key space to Token.*

1. Random Partitioner:
   - Tokens are integers in the range $0 - 2^{127}$.

    - MD5(key) -> Token.
    - Good: Even key distribution.
    - Bad: Inefficient range queries.

2. Order Preserving Partitioner:

    - Tokens are UTF8 strings in the range: "" - ∞.
    - Key -> Token.
    - Good: Efficient range queries.
    - Bad: Uneven key distribution.

2.5.3 Infinispan

The goal of this section is to explain the components and the operation of Infinispan. The content of this section is based on the keynote of the presentation [27]

Distributed in-memory key/value data grid and cache is the open-source version of JBoss. The system is composed of processes called nodes communicating via sockets. Each node has memory for storing data. Same data is replicated between multiple nodes. In this way the system can handle node failure while data is still available. It is also elastic, build on the idea that nodes can join or leave the system without affecting the clients.

For the "control path" JGroups is used [28] :

- Cluster creation and deletion. Cluster nodes can be spread across LANs or WANs
- Joining and leaving of clusters
- Membership detection and notification about joined/left/crashed cluster nodes
- Detection and removal of crashed nodes
- Sending and receiving of node-to-cluster messages (point-to-multipoint)
- Sending and receiving of node-to-node messages (point-to-point)

Infinispan can be either embedded in the same process with the application or clients can connect trough various remote access protocols located on different machines or clusters (REST API).

On top of the storage engine, there are different services such as: distributed transaction, queering, mapreduce tasks and monitoring.

High availability is ensured by replication. Because the memory is volatile, the same data is replicated to different nodes. Infinispan supports:

- Total replication: data is replicated to all nodes in the ring. This strategy is very fail-tolerant but the capacity is limited to the capacity of an individual node. The write speed is very low but the read speed is very high because you can read from any node with the guarantee that the same data is on each node. It can be used for storage of meta-data.

- Partial replication: data is distributed to some nodes called the owners of that data. This strategy is based on consistent hashing. If a owner fails, another node is choose and data is copied to that node.

2.6 Kafka: A high-throughput distributed messaging system

Kafka is a distributed, partitioned, replicated, multi-subscriber commit log.[29]

The project started at LinkedIn from a need of a monitoring system for events of their website such as: pageviews, keywords searched, etc. LinkedIn bring Kafka open-sourced project and is now used in many others projects. The systems that existed before Kafka had drawbacks like scaling and performance.

2.6.1 Descriptive of Kafka

The basic functionality of Kafka is message persistence. The Kafka system is composed of producers, kafka cluster and consumers. The Producers send the messages through the network to the Kafka Cluster. The messages are saved then the Kafka Cluster sends the messages through the network to the consumers.

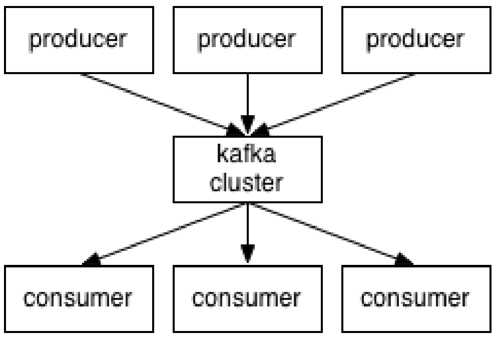


Figure 2.6: Basic image of Kafka.[30]

The messages are sorted by topic. Each topic is divided in partitions. Each partition contains an ordered sequence of the messages coming from producers(writers) (Figure 2.7). The messages are indexed by an id number(offset). In one partition, all id numbers are different. For each consumers, the broker manages the offset related to that consumer in order to know which message to read next. The offset is controlled by the consumer. By handling the offset, the consumer knows which message to retrieve from the broker.

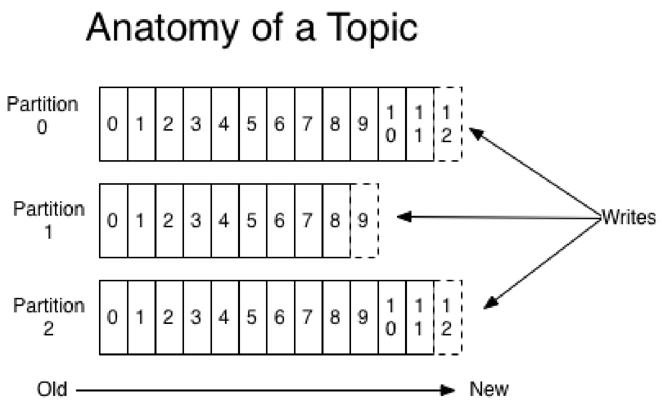


Figure 2.7: The anatomy of a topic.[30]

For reliability issues, each partition is replicated as follows: one leader manages all the read and write requests. Zero or many followers replicates the leader, without handling the read and write requests. In case of the leader crashes, a follower replaces the leader.

To summarize, Kafka guarantees[30]:

- Messages from producer are saved in order that they are sent.
- Consumers receive the messages in order that they are stored.
- If the factor of replication is N, then there can be N-1 server failures and the messages are not lost.

2.6.2 Trace of message in Kafka

This section explains how the messages are actually stored by looking at the source code[3] of Kafka. Kafka is a complete distributed system. We are interested to trace the message transit from its production until the message is consumed by focusing on how the message is stored by the broker.

A message is composed by a header and a payload. There are two ways to format a message (Figure 2.1). This allows different versions of message format to be used.

```
/**
* A message. The format of an N byte message is the following:
*
* If magic byte is 0
*
* 1. 1 byte "magic" identifier to allow format changes
* 2. 4 byte CRC32 of the payload
* 3. N - 5 byte payload
*
* If magic byte is 1
```

[3]kafka-0.8.0-beta1 Version.

```
*
* 1. 1 byte "magic" identifier to allow format changes
* 2. 1 byte "attributes" identifier to allow annotations on the message
   independent of the version (e.g. compression enabled, type of codec
   used)
* 3. 4 byte CRC32 of the payload
* 4. N - 6 byte payload
*/
```

Code 2.1: Message format.[31]

In addition to the format listed above, there is another field of 4 bytes which is used to store the length of the message.

For each partition of a topic (e.g. `my_topic`, Kafka creates directories named `my_topic_0`, `my_topic_1`, and so on. In each directory there are files[4] According to the Kafka documentation the offset starts at 00000000000. Each file has the name represented by the offset of the first message that it contains. The name of the first file is 00000000000.kafka. Let be X the offset of the file. The next file name will be a number between X and X + maximal size of a file. The next picture resume the structure of a log file.

[4]The size of files are configurable.

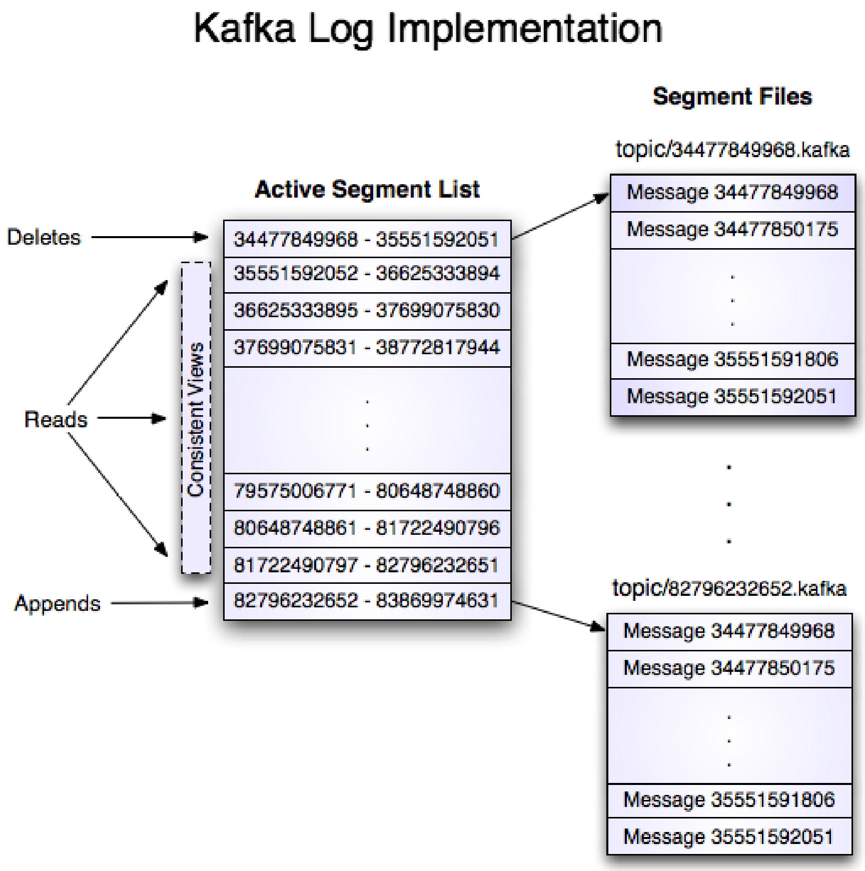


Figure 2.8: Kafka log implementation.[31]

In the Figure , the active segment list represents the topic directory where the file are stored. The consumers have access to messages in the consistent views space.

Example of message trace

The publisher uses a queue to store messages locally before sending it. This queue is as a local buffer for messages. To send the messages, the publisher use a thread that has access to the local queue. The thread and the queue runs in parallel. This allow the sending of a message to be independent of the producing of a message. The producer sends the messages to the broker in arrays of messages.

The message is received by the broker. The broker analyzes the header of the message in order to identify the topic. The 4 bytes of offset are added to the message. Afterwards, the message is stored in the topic partition where the publisher is linked.

2.6.3 Product quality analysis

The product quality model analysis is based on the ISO/IEC 25010 standard because is the most recent international standard expressing software quality in software engineering. More specifically, we consider the "product quality" because it decomposes the software quality into eight high level criteria. Among this criteria we choose the four following criteria because they target the message logging system:

Functional suitability: Functional completeness: Kafka is scalable, persistent and reliable.

Performance efficiency: Kafka is fast. *A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients.* [32]

Usability: Easy to learn and to deploy. Kafka has a good documentation.

Security: Kafka does not care about security at all. This is a drawback regarding the protection of messages.

Other criteria such as maintainability, portability and compatibility are considered out of the scope of this analysis.

2.7 Zper: ZMQ persistance broker

The section starts with the description of Zper. Next, the advantages and the inconveniences are discussed. Finally, the section ends with an analysis according to selected criteria belonging the ISO25010 standard.

ZPER is a ZeroMQ persistence broker.[33]

2.7.1 Description of Zper

In this section the components of the Zper are described. Zper is implemented in Java on top of the ZMQ framework.

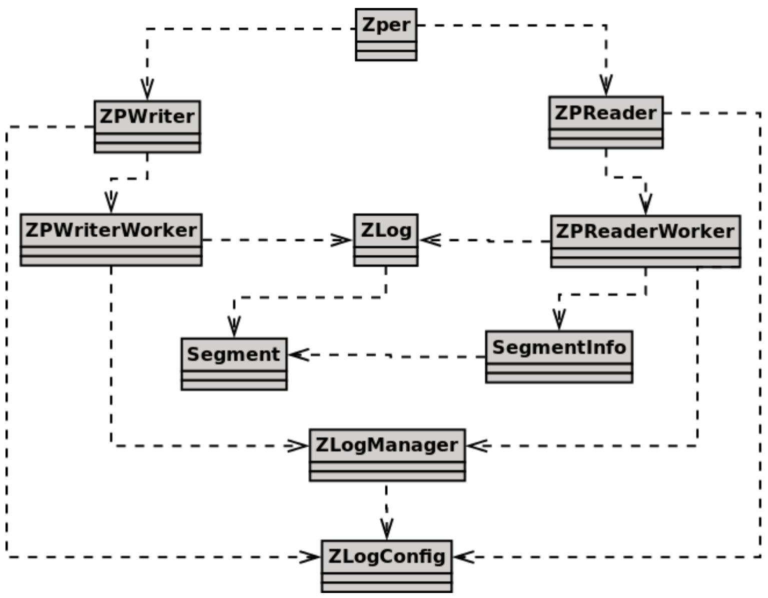


Figure 2.9: Zper Class diagram.

The class Zper contains the main method. When we execute this method, an instance of ZPWriter and an instance of ZPReader are created and launched, each in a separate thread. We are going to describer the writer and the reader processes separately. The writer processes will be described first.

The ZPWriter gets the configuration, read from the configuration file, as follow:

- *writer workers* is the number of writer workers, those that actually write messages on disk.
- *writer address* is the address and the port where the zper waits for clients to write messages.
- *base_dir* is the directory where the files will be saved.
- *segment_size* is the size of a file.
- *flush_messages* is a threshold of messages number; when this threshold is reached that means the messages should be written on the disk.
- *flush_interval* is a threshold in terms of time (milliseconds); when this threshold is reached that means the messages should be written on the disk.
- *retain_hours* is the number of hours the messages are kept on the disk.
- *recover* indicates if the recover options should be activated.

After the configuration, the ZPWriter launches the ZPWriterWorkers and routes all workers together thanks to the `inproc` transport protocol of ZMQ. Then, it waits for connections.

A ZPWriterWorker waits for messages. The format of the message is shown in Figure 2.10. After a message arrives, the ZPWriterWorker handles the message in four cases: `START, TOPIC, COUNT, MESSAGE`.

```
START case:
 + ---------- + --------- + ------------ + ----- + ------------- +
 |  1 byte-   |  1 byte-  |    1 byte-   | topic |    16 bytes    |
 |topic-hash  |    flag   | topic-length |       |      uuid      |
 + ---------- + --------- + ------------ + ----- + ------------- +

   Flags is: 0 (Fastest) : No response (DEALER and PUSH)
             1           : Response at flush (DEALER)
             2 (Slowest) : Response at every write (REQ)

TOPIC case: (the message is empty)
COUNT case:
   + ---------------------- +
   | 4 bytes-size of payload |
   + ---------------------- +
MESSAGE case:
   +-----------------------------------------------------------+
   | payload (the size is represented by the previous 4 bytes |
   +-----------------------------------------------------------+
```

Figure 2.10: Format of a message received by a ZPWriterWorker.[33]

A message arrives in several tranches. In the case `START`, ZPWriterWorker retrieves the flag and the topic from the message and asks to ZLogManager for the object ZLog corresponding to the topic retrieved. If the flag is 1 or 2, then, ZLogManager prepare a message for response. Then, it passed to the `COUNT` case where the message represents the size of the message. Then, it passes to the `MESSAGE` case where it retrieves the payload and calls the method appendBulk of the ZLog object retrieved before. The method appendBulk of ZLog writes the message to the current Segment.

The segments represents the files on disk. The files are structured like in Kafka, there are files of fixed size with the names represented by the offset.

Next, we focus on how the ZReader works. In the construction phase, ZReader gets the configuration from the configuration file.

- *base_dir* represents the directory where files are saved.
- *reader workers* is the number of readers.
- *reader address* is the address and the port where the clients will be connected.
- *send buffer* is the size of the buffer for sending the messages.

When the ZReader starts, it launches the ZReaderWorkers and routes them, then waits for connections. The ZReaderWorkers receive a request with the topic and a command (FETCH or OFFSET). After it treats the topic and gets the ZLog object corresponding to the topic, it analyses the command and process it. In case of the fetch command, the SegmentInfo is recupered from the ZLog object in order to get the path to the file. The file path and the offset are sent to the method process_file frocriteriim Persistance class that performs the read of the message from the file [34].

The main functionality of Zper is the message persistence. Zper is reliable. Depending of the flag used, the level of reliability is decided. In case of a failure, the subscribers can retrieve the last offset received. Zper is not scalable. The workers are configured when Zper is launched and cannot be dynamically adjusted.

2.7.2 Product quality analysis

In the same way as for Kafka, discussed in section 2.6.3, the product quality model analysis is based on the ISO/IEC 25010 standard.

Functional suitability: Functional completeness: Zper fulfills the persistence task. The persistence is the main characteristic of Zper.

Performance efficiency: In terms of time behavior, Zper is fast. Regarding the problem that it solves, this is an advantage for Zper.

Usability: Easier to use for ZMQ developers [33]. Zper has a poor documentation. However, Zper comes with a simple structure allowing users easily learn to operate it.

Security: Zper does not care about security at all. This is a drawback regarding the protection of messages.

Other criteria such as maintainability, portability and compatibility are beyond of this analysis.

2.8 Comparing Kafka and Zper systems

This section discusses the two systems presented before, Kafka in section 2.6 and Zper in section 2.7. Firstly, the advantages and the inconveniences of each system are introduced. Then, a table summarizes the comparison between product quality seen in section 2.6.3 for Kafka and in section 2.7.2 for Zper.

Both Kafka and Zper fulfil message persistence functionality. Kafka is scalable while Zper is not. Kafka is reliable in terms of failure through the replication feature. Zper can recover after a restart. In terms of performance, Zper is two times faster than Kafka [33]. Kafka is well documented. Neither, Kafka nor Zper do not pay attention for the authentication of different components or for the security of messages.

	Kafka	Zper
Functional suitability:	+++	- ++
persistent	✓	✓
scalable	✓	
reliable	✓	✓
Performance efficiency:	- + +	+++
time		✓
space	✓	✓
Usability:	+++	- ++
learn	✓	✓
documentation	✓	
Security:	- - -	- - -

Table 2.1: Comparison of Kafka and Zper systems.

According to the comparison table, Kafka is more suitable to be used in contexts where the reliability is important. Zper is more suitable to be used in contexts of high performance demand.

Chapter 3

Message Logging System

In this chapter, we explain the design of the Message Logging System. In the first section, the message persistence problem is described. In the second section, we address the problem and propose an architecture for the Message Logging System.

3.1 Problem space

The first goal is defined as: Store messages at Exchange level having a network I/O in a reliable way. The messages should be stored on the Exchange they arrive, in order to keep the latency low. The messages should also be stored on others Exchanges as to achieve reliability. In case an Exchange goes down, messages can be retrieved from the others Exchanges.

The second goal is defined as: Subscribers retrieve messages from Exchanges. In case an Exchange goes down, subscribers should be aware of the failure.

In what follows, the two mains goals are refined.

Store messages. In the RoQ elastic queue, the amount of messages is relatively important. This amount can reach the order of millions of messages per second. The size of a message can vary, but we hypothesized that it remains of order of Kbs. The messages are composed of bytes; we are not interested of the content of a message. A message has a topic and must be saved accordingly to the order of arrival.

A network I/O. The goal is to save messages on the Exchanges where they arrive.

Message persistence. Messages should be stored on disk. In case an Exchange goes down and restarts after the failure, the exchange should recover from the failure and find the previous messages.

Reliable system. The goal is to ensure the availability of messages even in case an Exchange downfalls. Messages should be stored on others Exchanges, in order to be retrieved by subscribers in case of need.

Easily retrieve messages. In case a subscriber wants to receive previous messages, the Exchange should be able to respond to his request.

Do not reduce the performance of the Exchange. Adding the property of storing messages implies more operations to be done at the Exchange's level. Those operations should not reduce the performance of the broker.

Do not create a bottleneck. Regarding the total throughput, the risk that a bottleneck occurs is very high.

3.2 Solution space

Considering the first goal, several existing storing system have been embedded with RoQ and tested the performances[1]. Data results of tests are in Appendix A. According to those results we propose a comparative table in order to identify the appropriate techniques to achieve the message logging goal.

	No-SQL	File System storage	Memory storage
Efficiency	- -+	-++	+++
Management	+++	-++	-++
Cost	- -+	-++	+++

Table 3.1: Storage possibilities of messages.

Considering the results of this table, we propose a solution that combines the Memory storage and the File System storage for the messages logging problem.

Firstly, the messages will be stored in the memory then they will be transfered on disk. By doing so, the messages remain available and can be immediately send as a response to a request coming from a client. In case the message is not available in the memory, MLS searches for the message on the disk.

To solve the reliability problem, a replication mechanisms is used. Messages are replicated to other Exchanges. For performance considerations regarding the transmission of messages between Exchanges, several messages grouped in a buffer are send through the zero-copy technique.

The File System storage is more convenient in our case, because the messages should be saved in order of arrival. According to results from Appendix A, it is more efficient to append a message to a file than to make an insert request to a No-SQL database.

[1]Six different configurations of Cassandra and one configuration of Infinispan were tested.

3.2.1 The architecture of Message Logging System

This section describes each component of the architecture, the behavior and the views of the system related to the mechanisms of logging the messages. The structure of one Exchange is based on the Publish/Subscribe [35] architectural pattern. Exchanges are connected between them like in a peer to peer network (P2P).

The Figure 3.1 gives a global view of the Message Logging System composed by three Exchanges.

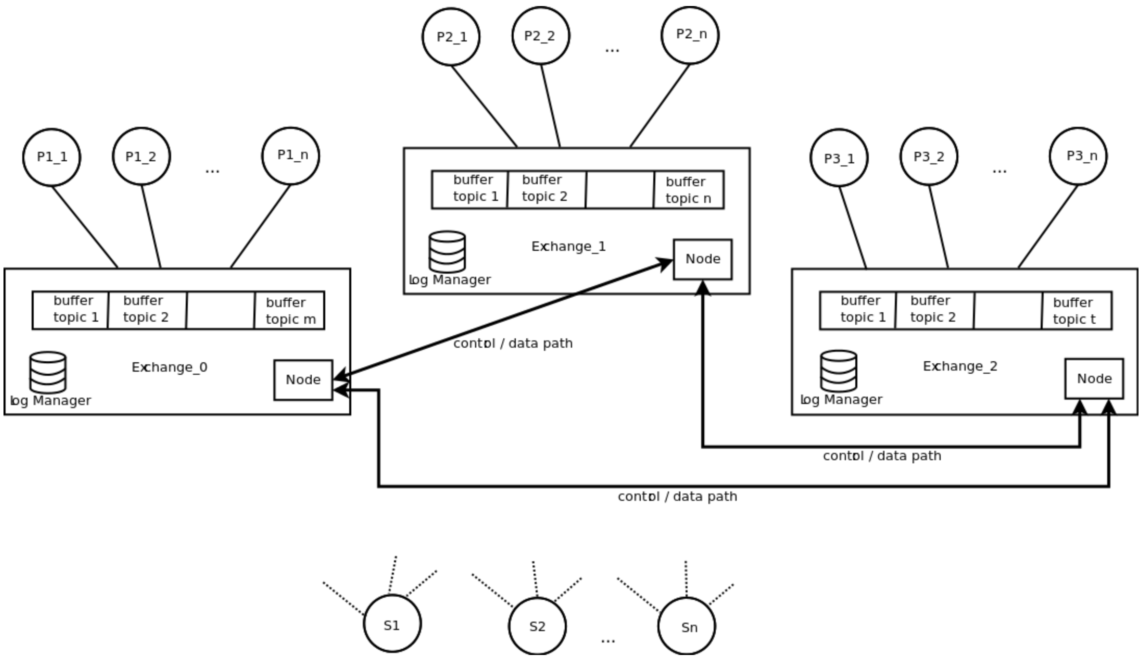


Figure 3.1: Message Log Management Architecture.

The Publishers are connected to Subscribers through an Exchange. Subscribers are connected to all Exchanges of RoQ. An Exchange keeps in memory a buffer for each topic of the messages received from the publishers.

Each Exchange have a control path and a data path. The control path is used to spread configuration information:

- announce the join of a new Exchange;
- announce the crash of an Exchange;
- decide which replica to choose.

The data path is used to replicates messages between active exchange(masters) and passive exchange(replicas).

Both control and data path are handled by the `Node` component. `Node` is embedded into Exchange.

En exchange must have at least one seed in order to announce his arrival. The list of all exchanges (list of seeds) is hold by the GCM. The Host ask GCM for the list before lunching an exchange. When an exchange starts, it connects to a seed and announce his arrival. The others spread the announce join of a new Exchange.

In what follow, we explain on a concrete example how exchanges interact with others.

Let be the replication factor 2.

The first exchange starts (let's say Exchange_0). There are no seeds. In this moment there are no replicas, but the messages which passes through Exchange_0 are saved locally on disk through the. The messages arrives in a buffer which is mapped to a file on the disk. The mapping and the storage is done through the Log Manager. In case the process Exchange_0 (JVM fails) fails, the buffer is still in memory and is handled (flushed to file) by the operating system.

Another exchange starts (Exchange_1). The seed is Exchange_0. Exchange_1 tell to Exchange_0 of his arrival. Exchange_0 knows the address of Exchange_1. Exchange_1 choose Exchange_0 to be his replica. Exchange_0 choose Exchange_1 to be his replica. They both starts sending/receiving messages from one to other through data path.

Another exchange starts (Exchange_2). The others exchanges Exchange_0 and Exchange_1 are announced of the arrival of Exchange_2. Exchange_2 choose to have replicas Exchange_0 and Exchange_1. On the other side, Exchange_0 and Exchange_1 needs another replica, so they both choose Exchange_2 to be their replica.

Others exchanges starts. Notice that each exchange have a number of others exchanges (masters) which replicates to him. We call this number, the number of masters. They will choose their replicas among of exchanges with the smallest number of masters.

How does the messages are stored and replicated?

The messages arrives to Exchange_0 and are stored to the buffer. When the buffer is full, the buffer is stored to the disk and is sent to Exchange_1 and Exchange_2. Meanwhile, others messages arrives on Exchange_0 and are saved to a second buffer. The second buffer is not send until the ack from Exchange_1 and Exchange_2 for the first buffer is not received by Exchange_0. When the acks are received, the second buffer is sent to Exchange_1 and Exchange_2 and the first buffer is used to store new arrivals messages. We expect that the time to get the ack is smaller than the time it takes to fill the buffer.

What happens when a node fails? (2 cases):

The exchange is a master. The buffer is handled by the Operation System and it is flushed to disk. Replicas of the exchange which failed detects the failure and announce the others exchanges. Replicas check if they are consistent (last message received by all replicas). All files are kept on each replica. In case that the crashed node restarts, it will recover the state before failure and connects to replicas.

The exchange is a replica. The master detects that one replica failed. The master pick another exchange to replicates data. The new exchange picked do not have previous messages. It will store all messages which arrives from the moment it was picked to be replica.

How does subscribers detect when an exchange fails?

Beside the publisher/subscriber pattern between exchanges and subscribers, each subscriber have a port to communicate with the exchanges.

When a master fails, replicas of that master will announce the failure of the master node and which topics were handled by that master exchange which failed. Each subscriber checks if it subscribed to some topics of the failed node. If so, it ask to replicas to send last messages. By doing so, the subscriber is able to check if it received all messages.

In what follow, we describe for each component the techniques used in receiving, storing and sending messages.

The Publisher

The role of the publisher is to send messages of the format shown in the Figure 3.2. The publisher is connected to an Exchange through a socket network. The messages are sent in an asynchronous mode.

```
+---------------------+-----------------------------------+
|   topic_of_message  |              message              |
+---------------------+-----------------------------------+
 \                  / \                                  /
         header                     payload
```

Figure 3.2: Format of a message sent by a publisher.

The Subscribe

The subscriber is connected to the all Exchanges through a socket network. The subscriber is registered by the Exchange and it receives the messages in the format shown in Figure 3.3. The subscriber can requests the Exchange to send him previous messages stored in the log.

```
+-----------------------------------+
|              message              |
+-----------------------------------+
```

Figure 3.3: Format of a message received by a subscriber.

The Exchange

The role of an Exchange is to store messages in memory (buffer) and to the Log Manager, then, to forward the messages (the buffer) to others Exchanges.

The Exchange has a buffer in the memory for each topic of messages. By using the buffer, the memory log technique is used in the design of the architecture. The trace of storing a message in the buffer is hereby reproduced:

- the message arrives from the publisher;
- the topic is identified by reading the header;
- the size of the message is computed;
- the size and the payload of the message are stored in the buffer.

Each buffer in the broker has the following structure:

- *buffer size* in bytes;
- *threshold* indicates the position in the buffer when it starts sending messages to the log;
- *last message index* points to the last byte of the last message saved in the buffer;
- *buffer space* represents the memory allocated for the buffer.

The messages are sent to others Exchanges by using the zero-copy technique.

The Log Manager

There are two roles the Log Manager accomplishes. The first one is to store in files the messages arriving from publishers. The second one is to respond at requests coming from subscribers. We are going to describe each role separately.

The Log Manager handles the files through a hierarchy shown in the Figure 3.4

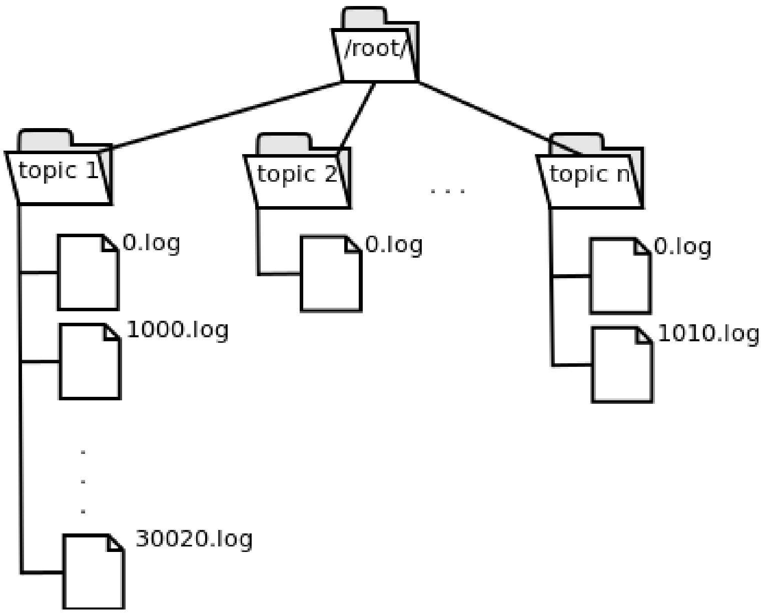


Figure 3.4: The files' hierarchy of the log.

Each topic is represented by a directory named after the topic name. Each directory contains files of recommended[2] size, named according to the following strategy:

- The name of the first file is `0.log`
- The following files are named in accordance with the total size in bytes of messages received.

For example, in Figure 3.4 in the directory `topic n`, we are in the first file, called `0.log`. If the recommended size for a file is 1000 bytes and the file is filled up to 980 bytes then the next message arriving, which has 30 bytes length, is saved to the file that has now a size of 1010 bytes. The name of the next file will be `1010.log`

The log manager keeps a file opened for each topic. The messages are appended to the end of the file of the respective topic.

The second role of the Log Manager is to respond at subscribers requests. The Log Manager is able to find a file of a topic inside an Exchange and to find a particular message inside a file.

In the design of the Log Manager we have used the zero-copy technique. Neither Kafka nor Zper [34] do not use zero-copy. The zero-copy technique allows preserving a good performance when the number of the message transmitted is significantly large.

The motivation for having the names of the files accordingly of the offset comes from two sources. Firstly, it is easier to identify a file by the offset, in order to retrieve a message of a certain offset from that file. Secondly, the offset type is a

[2]The size can vary accordingly to the last message that completes the file.

long, which is significantly large. The maximum number that can be represented by a long is 9,223,372,036,854,775,807 [36]. Let's say 50 Mb/s is the rate at which a file is written and the offset is represented by a long. This means it will reach the maximum of the offset in more than 100.000 years.

Chapter 4

Integration of Message Logging System into RoQ

This chapter is divided in three sections. The first section describes how the architecture presented in the previous chapter is implemented. In the second section the description of Messages Storage implementation. The third section describes the Node Communication implementation.

4.1 Implementation of Message Logging System

This section explains the integration of the Message Logging System into the RoQ system. A global view of the Message Logging System is described in Figure 4.1.

The implementation of the module is made in Java. We use the ZMQ library for the network communication between the components. The Publishers and the Subscribers are connected to the Exchange. We are focusing on the Message Logging System, therefore, only the components of the Message Logging System are illustrated in the Figure 4.1 and explained afterwards.

When the Exchange is launched, it creates an instance of class `MessagePersistence`. Inside the constructor of MessagePersistence two objects are created. The first object `storage`, instance of class `MessageHandler`, is responsible for storage messages of this Exchange. The second object `node`, instance of class `Node`, is created in a separate thread and is responsible for the communication with others Exchanges. The configuration data needed to create the topology of the system, to announce the join of a new Exchange or the failure of an Exchange also the heartbeat is handled through this object `node`. The buffers are sent through the same object `node`.

The Exchange class The constructor receives in parameters a String which represent the addresses to be used by the `node` object, a String which represent a list of seed addresses of others Exchanges and a Sting representing the path through the configuration file.

After the Exchange is launched, in the while(true) loop of the `run` method, each time a message comes from publishers the method `handle` of the class MessageHandler is called. This method takes in parameters the topic of type String, the ID of the publisher of type String and the message of type byte[].

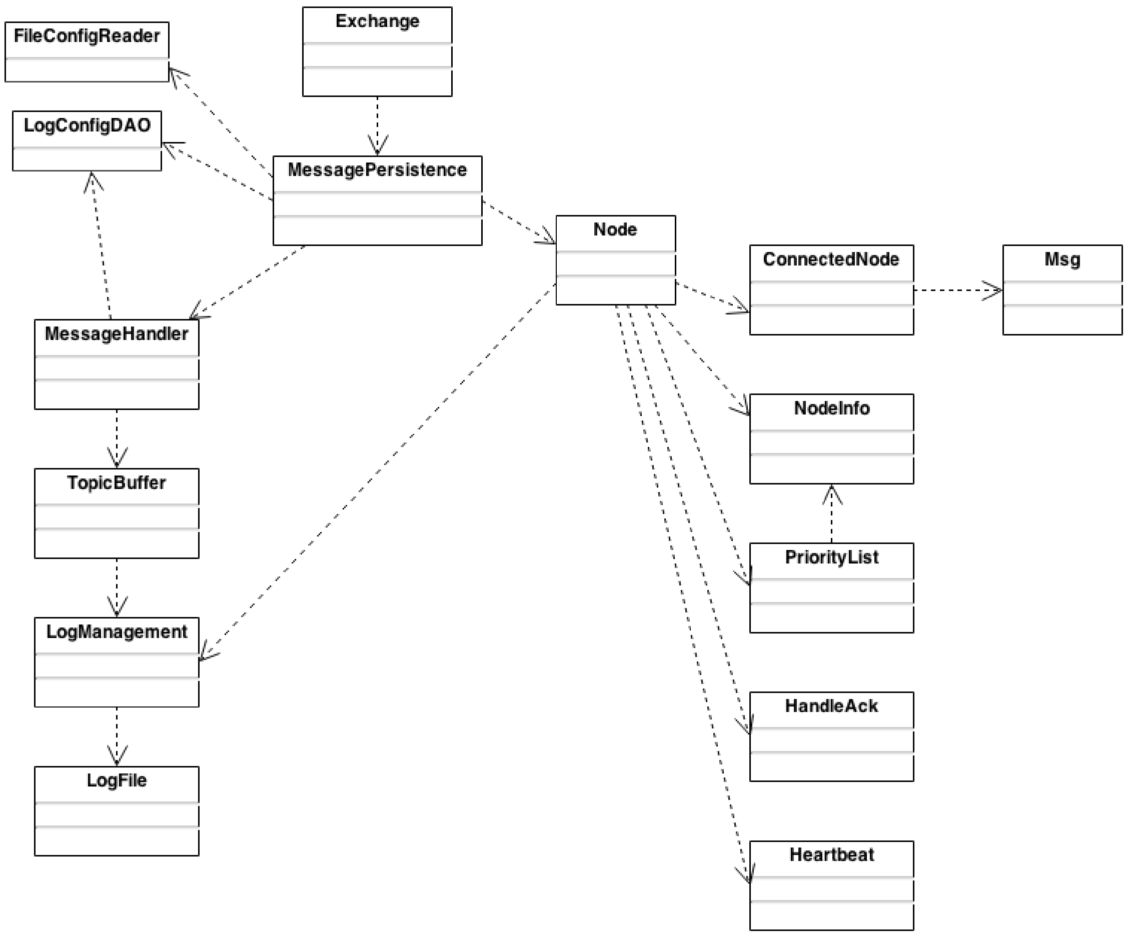


Figure 4.1: Class diagrams.

The next two sections describe each class of the two sides. Firstly the storage of messages "locally" and secondly the communication between Exchanges.

4.2 Messages Storage implementation

The MessageHandler class The constructor MessageHandler connects to the Node through a ZMQ Socket of type `inproc`.

The exchange's buffer is kept in a HashMap. The key is the topic and the value is an object of type TopicBuffer. The buffer hierarchy is shown in Figure 4.2. The MessageHandler class contains the method `manage(topic, message)` which is used for saving a message in the buffer. When a message of a new topic arrives, then the method `createTopicBuffer(topic)` is called. In order to retrieve a message from the buffer, the method `loadFromBuffer(topic, offset)` is called.

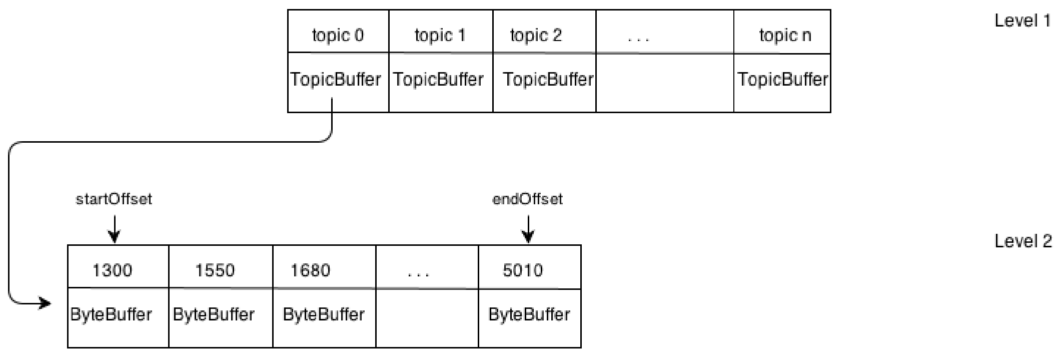


Figure 4.2: Exchange Buffer hierarchy.

The saving of a message and the retrieval of a previously saved message from the buffer are now described:

- put message to buffer
  - handle method is called with a topic of type String, the ID of publisher of type String, and a message of type byte[].
  - checks if the topic exists and it creates a new one if it does not exists.
  - creates a ByteBuffer and puts the message in it. The ByteBuffer is an object from the java.nio package. We have to use this object in order to be able to use the zero-copy method.
  - add the message to the buffer by calling the putMessage(ByteBuffer message, Socket inproc, String nodeName) method of the TopicBuffer class. The putMessage method puts at the end of the Level 2 buffer with the key endOffset the message received in parameter (see Figure 4.2).
- get message from buffer
  - loadFromBuffer method is called with a topic of type String and a offset of type long.
  - if exists, get the TopicBuffer from the exchange buffer (see Figure 4.2 level 1).
  - if exists, the message is retrieved from the TopicBuffer (level 2) by asking the value at the key represented by offset.

The TopicBuffer class represents a data structure composed by HashMap<Long, ByteBuffer> buffer, two indexes startOffset and endOffset of type long and a bufferToSend of type ByteBuffer. There are two buffers in this class:

The buffer HashMap structure works like a queue. The messages are saved at the end of the buffer, in the element pointed by the value of the endOffset.

The bufferToSend is used for sending multiple messages at one time to the Log Manager. The bufferToSend has a fixed size of 65000 bytes and, in order to be sent through zero-copy method, it should be created by the method `allocateDirect(buffersize)`

[37]. For reliable propose the bufferToSend is of type MappedByteBuffer. The next protocol describes how a message is saved in the buffer in order to send the whole buffer at one time to the concerned file and to others Exchanges through `node` object. This buffer is used for performance reasoning because it is better to use the sendZeroCopy method with large messages. [8]

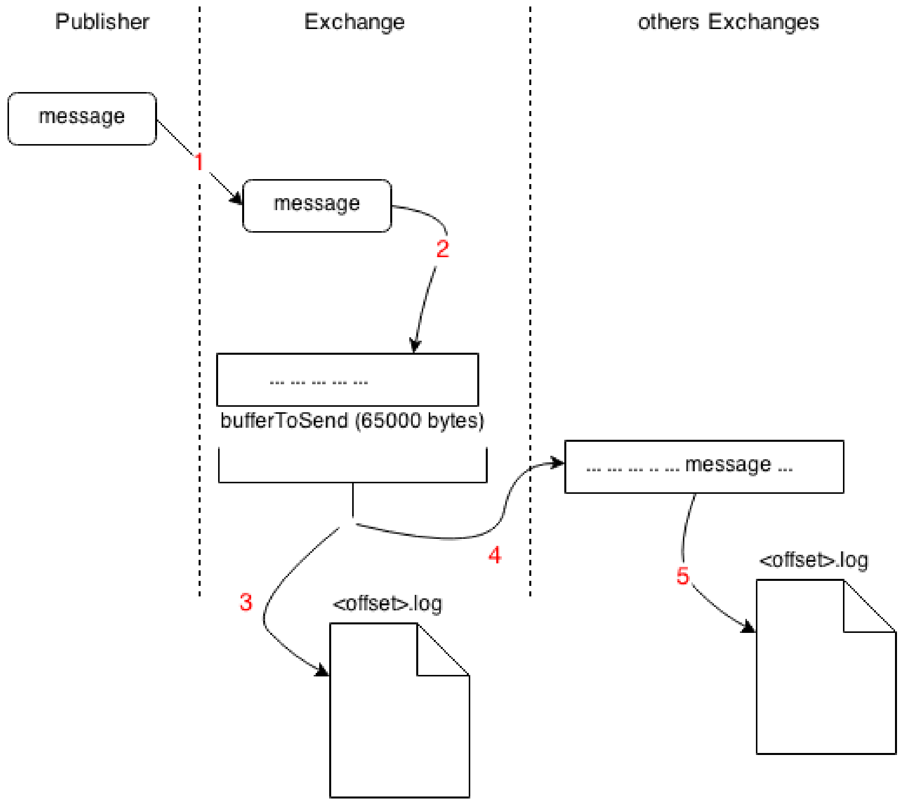


Figure 4.3: The bufferToSend protocol.

(1) the message arrives from the publisher.

(2) the size of the message and the message are saved in the buffer.

(3) store the buffer locally

(4) sends the buffer to the others Exchanges.

(5) the buffer is received by the others Exchanges and it is saved into the concerned file.

Two special cases can occur: either the arrived message is larger than the remaining space in the buffer; or the arrived message is larger than the buffer itself. In order to handle these cases, a new algorithm is developed.

(a) the message arrives from publisher.

(b) check if there is enough space to put the message and his size represented by a int.

    - if there is enough space, put the size (4 bytes) and the messages, go to step (a).
    - if in the buffer has not enough space, store the buffer and then send the buffer to the others Exchanges.
    - check if the size of the message + 4 bytes is larger then the size of the buffer.
        * if true, create a new buffer of the size of the message + 4, add the size of the message in the beginning of the new buffer and add the message, then store the buffer and send the new buffer to the others Exchanges.
        * if false then go to step (b).

The LogFile class is a data structure composed by a ArrayList<long>, FileOutputStream, fileSize, startOffset and offset.

- *ArrayList<long>* is a list of file names which exist on the disk for a certain topic.
- *FileOutputStream* is the current opened file.
- *fileSize* represents the size of the current file.
- *startOffset* is the offset of the first message in the file, it represents also the name of the file.
- *offset* represents the actual offset of the current file.

The LogManagement class handles the hierarchy of files saved on the disk. The LogManagement class has a HashMap<String, LogFile> as attribute. This hash map allows to easily associate a LogFile to each topic.

Storing a message:

- First, it checks if the topic exists.
    - if it does not exist, then creates a file named *0.log*
    - if it does exist, then checks whether the size of current file exceeds the maxFileSize, if it is the case, then create a new file named *<value of offset>.log*
- updates the offset and the size of the file.
- stores the size of the message and the message itself in the file.

4.3 The Node communication implementation

The Node class handles the communication with others Exchanges. It runs in a separate thread than the Exchange in order to receive and send configuration information and buffers of messages while the Exchange receives messages from publishers.

Through the seeds received at the creation of the Exchange, the node receives information, on the configuration path, about the system topology.

The node choose replicas by asking the PriorityList which are the low charged nodes.

In the while(true) of the `run` method, messages are sent to and received from others Exchanges. Messages of type configuration are:

- heartbeat - from others connected Exchanges every two seconds.
- ack - from replica Exchange when a buffer was received.
- announce a crash - to others connected Exchanges when a crash is detected or announced by another Exchange.
- announce a join of a new Exchange - to others connected Exchanges.
- ask for list of Exchanges - to the Exchange which asked for it.

Beside the configuration messages, there are messages which contains buffers to be send or received.

The ConnectedNode class contains information about a connected Exchange such as: the name, configuration Socket, data Socket, the value of the last heartbeat and a list of messages which waits to be acknowledged.

The NodeInfo class contains information about an Exchange: name, address, configuration port, data port.

Chapter 5

Case studies and evaluation

This chapter is dedicated to analyzing the overall performances of the RoQ system before integrating the Message Logging System and after integration. We have tested the application on Amazon EC2 virtual servers in a network of three machines. Amazon EC2 is Amazon's Elastic Compute Cloud web service that provides resizable cloud computing capacity. It is intended to facilitate web-scale computing. The Global Controller Manager (GCM) is launched on the first server, while the a Host Controller Manager (HCM1), respectively another Host Controller Manager (HCM2), are launched on the second, respectively, the third server.

All servers share the same configuration:

- CPU :
  - product: Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz
  - vendor: Intel Corp.
  - version: 6.13.7
  - size: 1800MHz
  - width: 32 bits
- RAM : size 851 MiB
- Network :
  - description: Ethernet interface
  - configuration: broadcast=yes driver=vif link=yes multicast=yes
- OS: Ubuntu raring 13.04 i386 server
- Java 1.7, ZMQ 3.2.3 and JZMQ 2.2.0

Here are the configuration files for each component of RoQ:

```
#Global configuration management properties

#SECTION 1 FREQUENCY
period=60000
```

```
#SECTION 2 Server configuration
#Define whether we need to format the DB on startup
formatDB=false
```

Code 5.1: GCM.properties [38]

```
#Host configuration management properties

#SECTION 1 The global configuration manager server
gcm.address=172.31.0.203

#SECTION 2 IP address registration
#Define whether need to use specific network interface to connect
# Un comment this if you want to use
#network.interface =vbr0

#SECTION 3 Queue Creation configuration
#Exchange base port
exchange.base.port = 6000
#Monitor base port
monitor.base.port = 5500
#Stat Monitor base port
statmonitor.base.port = 5800
#Max number of exchanges per host (used in auto scaling)
exchange.max.perhost = 3
#Define whether we create the queue on 1 VM or each element (monitor,
#Scaling process, stat monitor) in the same VM as the Host configuration
   manager
queue.hcm.vm= true
#Define whether we must create the exchanges on the same VM
#as the HCM or in its own process
exchange.hcm.vm = false
#The memory heap of the JVM used to start the Exchanges
#This property is only used when the exchange.hcm.vm = false
exchange.vm.heap= 512

#SECTION 4 Statistic for monitor
monitor.stat.period =60000

#SECTION 5 Message Log configuration
directory=/home/ubuntu/master/log/
maxFileSize=5000000
duration = 5
replicationFactor = 2
useLog=true
```

Code 5.2: HCM.properties [38]

5.1 Message Logging System performances

The goal of this example is to see what is the impact of in terms of performances by integrating the Message Logging System module in RoQ. We are going to vary the number of messages until will reach the maxim capacity of Exchanges.

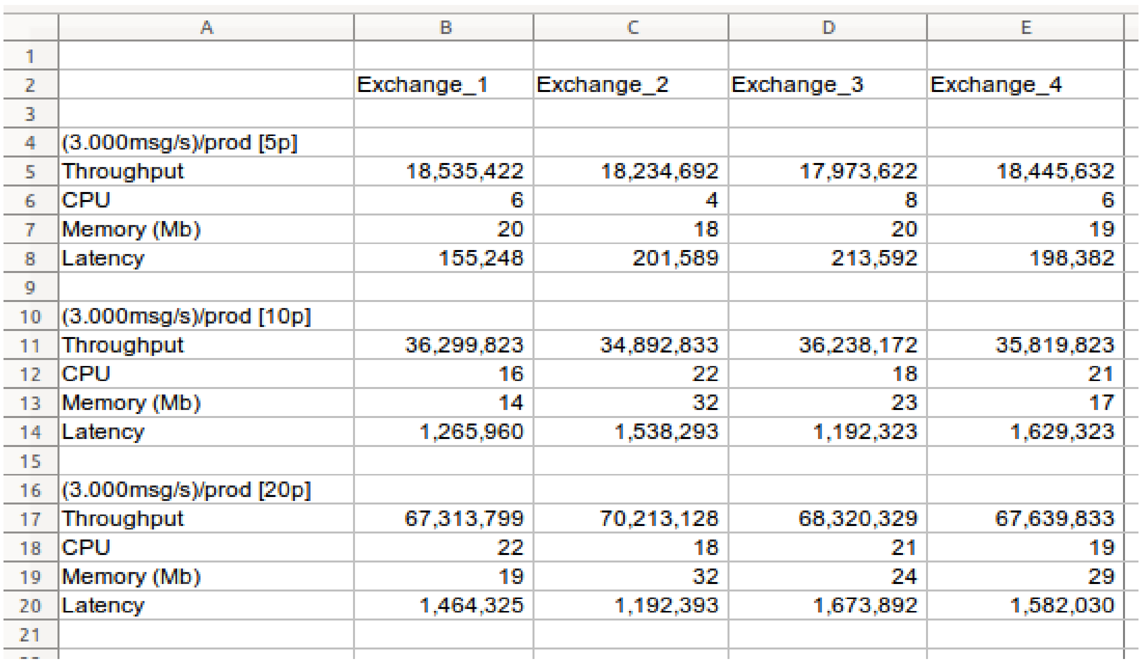
The system is composed of two host. Each host handle two exchanges. The replication factor is two. The number of messages/second per producer is the same for all testes (3000msg/s per producer). The number of producers varies: 5, 10 and 20 producers.

The next three calculus give an image of the value of the throughput that should be expected in tests:

$5prod * 3.000 = 15.000 msg/s;$
$20 * 15.000 = 300.000 bytes/sec;$
$60 * 300.000 = 18.000.000 bytes/min$

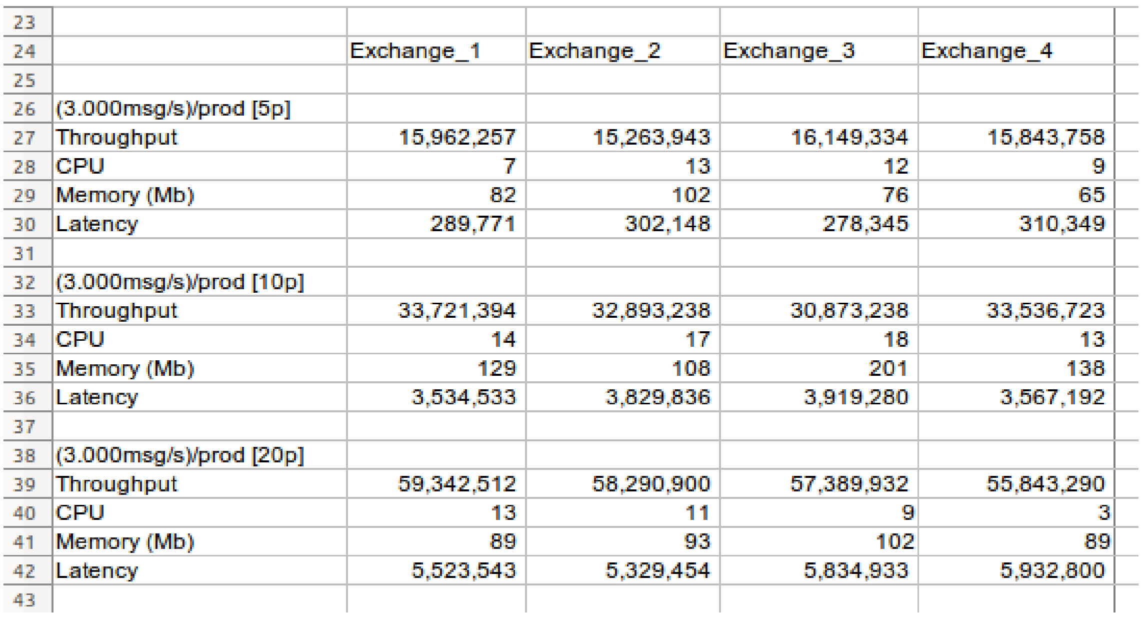
$10prod * 3.000 = 30.000 msg/s;$
$20 * 30.000 = 600.000 bytes/sec;$
$60 * 600.000 = 36.000.000 bytes/min$

$20prod * 3.000 = 60.000 msg/s;$
$20 * 60.000 = 1.200.000 bytes/sec;$
$60 * 1.200.000 = 72.000.000 bytes/min$



	A	B	C	D	E
1					
2		Exchange_1	Exchange_2	Exchange_3	Exchange_4
3					
4	(3.000msg/s)/prod [5p]				
5	Throughput	18,535,422	18,234,692	17,973,622	18,445,632
6	CPU	6	4	8	6
7	Memory (Mb)	20	18	20	19
8	Latency	155,248	201,589	213,592	198,382
9					
10	(3.000msg/s)/prod [10p]				
11	Throughput	36,299,823	34,892,833	36,238,172	35,819,823
12	CPU	16	22	18	21
13	Memory (Mb)	14	32	23	17
14	Latency	1,265,960	1,538,293	1,192,323	1,629,323
15					
16	(3.000msg/s)/prod [20p]				
17	Throughput	67,313,799	70,213,128	68,320,329	67,639,833
18	CPU	22	18	21	19
19	Memory (Mb)	19	32	24	29
20	Latency	1,464,325	1,192,393	1,673,892	1,582,030
21					

Figure 5.1: Tests data results of RoQ.



	Exchange_1	Exchange_2	Exchange_3	Exchange_4
(3.000msg/s)/prod [5p]				
Throughput	15,962,257	15,263,943	16,149,334	15,843,758
CPU	7	13	12	9
Memory (Mb)	82	102	76	65
Latency	289,771	302,148	278,345	310,349
(3.000msg/s)/prod [10p]				
Throughput	33,721,394	32,893,238	30,873,238	33,536,723
CPU	14	17	18	13
Memory (Mb)	129	108	201	138
Latency	3,534,533	3,829,836	3,919,280	3,567,192
(3.000msg/s)/prod [20p]				
Throughput	59,342,512	58,290,900	57,389,932	55,843,290
CPU	13	11	9	3
Memory (Mb)	89	93	102	89
Latency	5,523,543	5,329,454	5,834,933	5,932,800

Figure 5.2: Tests data results of RoQ with Message Logging System.

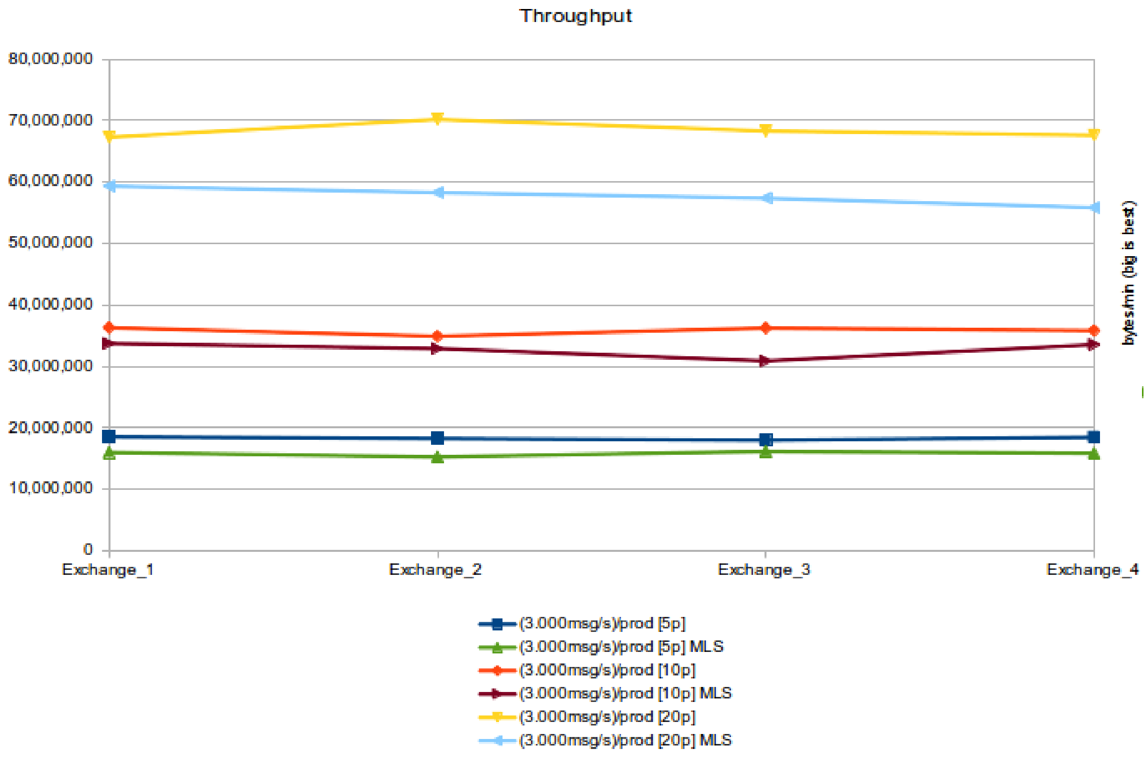


Figure 5.3: Graphic based on two previous tables representing the throughput.

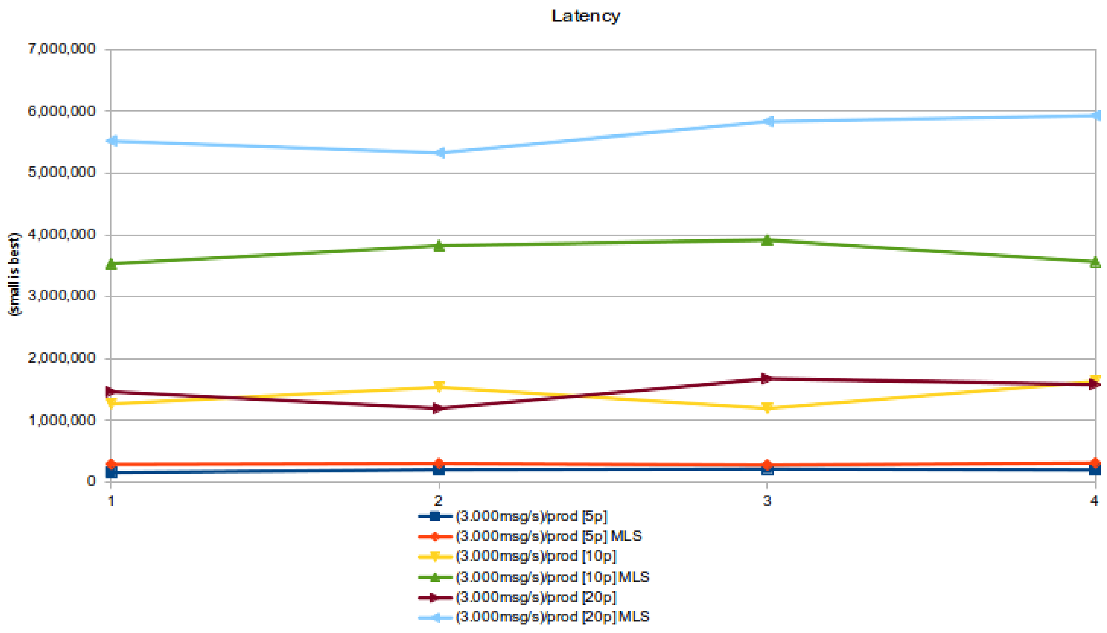


Figure 5.4: Graphic based on two previous tables representing the latency.

The throughput varies between the two systems with an advantage for the RoQ system without MLS. As for the latency. The difference in terms of performance between the two systems is expected. The operations added with the Message Logging System consist in handling messages into memory, writing them to files and sending to others Exchanges. Those operations reduce the throughput and increase the latency of the system.

Chapter 6

Conclusion and Future work

This chapter contains three sections. In the first section a summery of third and forth chapters representing the original contribution of this thesis. The second section discusses the advantages and shortcomings of the design and implementation of the architecture. Finally, the last section is dedicated to the short- and long- future work, based on the discussion of advantages and shortcomings.

6.1 Summary of the contribution

In the third chapter 3 an architecture for a Message Logging System is proposed. This architecture is designed in order to allow an Exchange to save a big number of messages and to easily retrieve those messages. To respond at those constraints, two techniques were used. First, the memory log technique seen in section 2.2 represented be a system of buffers in memory on the Exchange side section 4.2. This technique allow Exchanges to easily respond at request for messages coming from the Subscribers. Second, the zero-copy technique seen in section 2.1 which allows fast transfer of messages from the Exchange to the others Exchanges. The design of the architecture allows to handle the cases where a new Exchange appears or an existing Exchange fails, without loosing messages.

In the forth chapter 4 an implementation of the architecture is proposed for the RoQ system. The implementation is done in Java and use the ZMQ library for socket communication. This library supports zero-copy technique. For performance reasoning the messages are grouped in a buffer before sending them, see figure 5.4.

6.2 Advantages and shortcomings

The Message Logging System makes the RoQ system a persistent one. The persistence is the main advantage coming with the Message Logging System. The subscribers are able to retrieve messages on demand at any time.

Another advantage is that the Message Logging System do not influence any RoQ property. It is still elastic, the exchanges can disconnect without loosing any messages. New exchanges can join the system. The names of exchanges and the topics of messages are used to create the files hierarchy. In case an Exchange fails and restarts, it can recover the files and continue adding new messages to those files.

Regarding to the results of testing conducted in the fifth chapter (**??**), no significant performance reduction when using Message Logging System was observed.

A shortcoming is that the zero-copy from ZMQ library is bugged and related to the [39] the method does not perform zero-copy. However, it is on the issue list to be solved by the JZMQ developer team.

6.3 Future work

In this section firstly, problems to be solved in short term are identified. Secondly, a problem considered to be solved on long term.

6.3.1 Short term

Firstly, allow subscribers to retrieve more messages one time. Secondly, find a solution for the problem of zero-copy method in the ZMQ library. The zero-copy problem can be solved by using transferTo method from FileChannel of nio java package.

6.3.2 Long term

Increase the performance by allowing more buffers to be send while replication and change the acknowledgement system to be more flexible.

Bibliography

[1] Maarten Van Steen Andrew S. Tanenbaum. *Distributed systems: principles and paradigms*. Pearson Prentice Hall, 2 edition, 2006.

[2] Eric Brewer. Towards robust distributed systems. *Symposium on Principles of Distributed Computing (PODC) Keynote*, 2000.

[3] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[4] Roq wiki. `https://github.com/roq-messaging/RoQ/wiki`. Accessed: 2013-07-21.

[5] Nam-Luc Tran, Sabri Skhiri, and Esteban Zimányi. Eqs: An elastic and scalable message queue for the cloud. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 391–398, Washington, DC, USA, 2011. IEEE Computer Society.

[6] Oracle. Message-oriented middleware (mom). `http://docs.oracle.com/cd/E19148-01/819-4470/gbpdl/index.html`. Accessed: 2013-08-05.

[7] Cisco. Configuring system message logging. In *Cisco Fabric Manager System Management Configuration Guide*. Cisco Systems, San Jose, 2010.

[8] Sathish K. Palaniappan and Pramod B. Nagaraja. Efficient data transfer through zero copy. *IBM developerWorks*, 2008.

[9] Liu Tianhua, Zhu Hongfeng, Chang Guira, and Zhou Chuansheng. Research and implementation of zero-copy technology in linux. 2006.

[10] Jeffrey Dutky. Memory log. `http://www.dutky.info/jeff/software/mlog/`. Accessed: 2013-07-30.

[11] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. Buffer-on-board memory systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 392–403, Washington, DC, USA, 2012. IEEE Computer Society.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[13] Pieter Hintjens. *Code Connected: Learning ZeroMQ*, volume 1. iMatix Corporation, 2013.

[14] Pieter Hintjens. Zmq guide. `http://zguide.zeromq.org/page:all`. Accessed: 2013-08-05.

[15] W. Sliwinski A. Dworak, F. Ehm and M. Sobczak. Middleware trends and market leaders 2011, 2011. CERN, Geneva, Switzerland.

[16] EuraNova. Roq - elastic messaging for the cloud. `http://roq-messaging.org/`. Accessed: 2013-08-15.

[17] EuraNova. Roq documentarion. `https://github.com/roq-messaging/RoQ/wiki/Documentation`. Accessed: 2013-07-30.

[18] Andrew Brampton, Andrew MacQuire, Idris A. Rai, Nicholas J. P. Race, and Laurent Mathy. Stealth distributed hash table: A robust and flexible superpeered dht. In *Proceedings of the 2006 ACM CoNEXT Conference*, CoNEXT '06, pages 19:1–19:12, New York, NY, USA, 2006. ACM.

[19] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Robert E. Gruber Fay Chang, Jeffrey Dean. Bigtable: A distributed storage system for structured data. *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

[20] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[21] Ricky Ho. Hbase operation. `http://horicky.blogspot.be/2010/10/bigtable-model-with-cassandra-and-hbase.html`. Published: 2013.

[22] Cassandra site. `http://cassandra.apache.org/`. Accessed: 2013-12-10.

[23] Hans-Arno Jacobsen Sergio Gomez-Villamor Victor Muntes-Mulero Serge Mankovskii Tilmann Rabl, Mohammad Sadoghi. Solving big data challenges for enterprise application performance management. 2012-08-27.

[24] Apache Cassandra. Datastax site. `http://www.datastax.com/what-we-offer/products-services/datastax-enterprise/apache-cassandra`. Accessed: 2013-12-10.

[25] Srinath Perera. Consider the apache cassandra database. `http://www.ibm.com/developerworks/library/os-apache-cassandra/`. July, 03 2012.

[26] Benjamin Black. Cassandra, replication and consistency. `http://fr.slideshare.net/benjaminblack/introduction-to-cassandra-replication-and-consistency`. Keynote, 2010-04-28.

[27] Mircea Markus. Infinispan - the open source data grid platform (keynote). : 2013-12-22.

[28] Bela Ban. Jgroups. `www.jgroups.org`. Accessed: 2013-12-22.

[29] Kafka desgin. `http://kafka.apache.org/design.html`. Accessed: 2013-07-20.

[30] Kafka introduction. `http://kafka.apache.org/introduction.html`. Accessed: 2013-07-20.

[31] Kafka implementation. `http://kafka.apache.org/implementation.html`. Accessed: 2013-07-21.

[32] Kafka. `http://kafka.apache.org`. Accessed: 2013-08-16.

[33] Zper. `https://github.com/miniway/zper`. Accessed: 2013-07-21.

[34] Discution with Dongmin Yu, the autor of the zper.

[35] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermerrec. The many faces of publish/subscribe. *ACM Computing Surveys*, Vol. 35(No. 2):114–131, June 2003.

[36] Oracle. Primitive data types. `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`. Accessed: 2013-07-05.

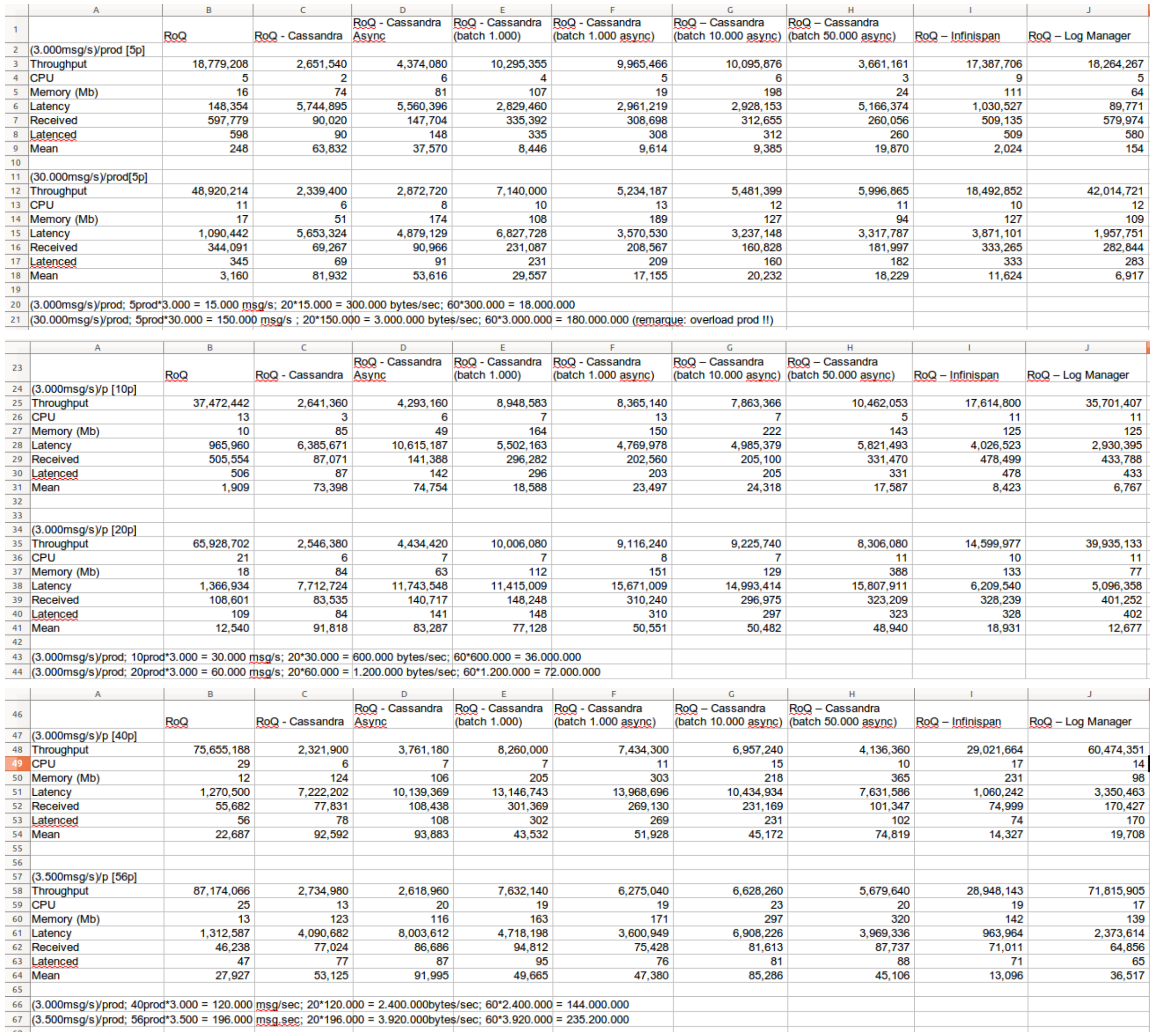
[37] Kenneth L. Calvert and Michael J. Donahoo. *TCP/IP Sockets in Java: Practical Guide for Programmers*. Morgan Kaufmann Pub, 2 edition, 8 2011. Pages: 123-124.

[38] Roq configuration files. `https://github.com/roq-messaging/RoQ/wiki/Configuration-files`. Accessed: 2013-07-21.

[39] Trevor Bernard. Zeromq - jzmq. `https://github.com/zeromq/jzmq/issues/189`. Date: March 2013.

Appendix A

This appendix contains data results from tests of RoQ with Cassandra and Infinispan. The tests are done on the local machine, on one Exchange. In the first column are the results of RoQ without any changes. In the next 6 columns are the results of RoQ with Cassandra (different configurations). Cassandra is embedded in the Exchange. Next, Infinispan is tested. Infinispan store the messages only in memory not on the disk, this is the reason that the latency is so small. In the last column are the results of RoQ with Log Manager



	RoQ	RoQ - Cassandra	RoQ - Cassandra Async	RoQ - Cassandra (batch 1.000)	RoQ - Cassandra (batch 1.000 async)	RoQ – Cassandra (batch 10.000 async)	RoQ – Cassandra (batch 50.000 async)	RoQ – Infinispan	RoQ – Log Manager
(3.000msg/s)/prod [5p]									
Throughput	18,779,208	2,651,540	4,374,080	10,295,355	9,965,466	10,095,876	3,661,161	17,387,706	18,264,267
CPU	5	2	6	4	5	6	3	9	5
Memory (Mb)	16	74	81	107	19	198	24	111	64
Latency	148,354	5,744,895	5,560,396	2,829,460	2,961,219	2,928,153	5,166,374	1,030,527	89,771
Received	597,779	90,020	147,704	335,392	308,698	312,655	260,056	509,135	579,974
Latenced	598	90	148	335	308	312	260	509	580
Mean	248	63,832	37,570	8,446	9,614	9,385	19,870	2,024	154
(30.000msg/s)/prod[5p]									
Throughput	48,920,214	2,339,400	2,872,720	7,140,000	5,234,187	5,481,399	5,996,865	18,492,852	42,014,721
CPU	11	6	8	10	13	12	11	10	12
Memory (Mb)	17	51	174	108	189	127	94	127	109
Latency	1,090,442	5,653,324	4,879,129	6,827,728	3,570,530	3,237,148	3,317,787	3,871,101	1,957,751
Received	344,091	69,267	90,966	231,087	208,567	160,828	181,997	333,265	282,844
Latenced	345	69	91	231	209	160	182	333	283
Mean	3,160	81,932	53,616	29,557	17,155	20,232	18,229	11,624	6,917

(3.000msg/s)/prod; 5prod*3.000 = 15.000 msg/s; 20*15.000 = 300.000 bytes/sec; 60*300.000 = 18.000.000
(30.000msg/s)/prod; 5prod*30.000 = 150.000 msg/s ; 20*150.000 = 3.000.000 bytes/sec; 60*3.000.000 = 180.000.000 (remarque: overload prod !!)

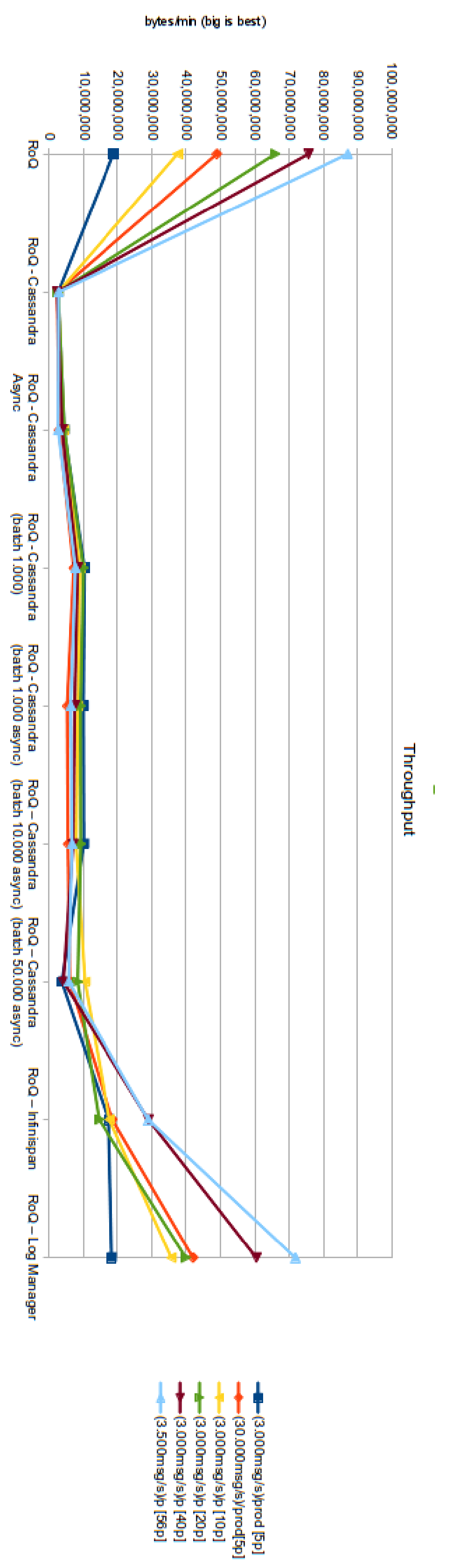
	RoQ	RoQ - Cassandra	RoQ - Cassandra Async	RoQ - Cassandra (batch 1.000)	RoQ - Cassandra (batch 1.000 async)	RoQ – Cassandra (batch 10.000 async)	RoQ – Cassandra (batch 50.000 async)	RoQ – Infinispan	RoQ – Log Manager
(3.000msg/s)/p [10p]									
Throughput	37,472,442	2,641,360	4,293,160	8,948,583	8,365,140	7,863,366	10,462,053	17,614,800	35,701,407
CPU	13	3	6	7	13	7	5	11	11
Memory (Mb)	10	85	49	164	150	222	143	125	125
Latency	965,960	6,385,671	10,615,187	5,502,163	4,769,978	4,985,379	5,821,493	4,026,523	2,930,395
Received	505,554	87,071	141,388	296,282	202,560	205,100	331,470	478,499	433,788
Latenced	506	87	142	296	203	205	331	478	433
Mean	1,909	73,398	74,754	18,588	23,497	24,318	17,587	8,423	6,767
(3.000msg/s)/p [20p]									
Throughput	65,928,702	2,546,380	4,434,420	10,006,080	9,116,240	9,225,740	8,306,080	14,599,977	39,935,133
CPU	21	6	7	7	8	7	11	10	11
Memory (Mb)	18	84	63	112	151	129	388	133	77
Latency	1,366,934	7,712,724	11,743,548	11,415,009	15,671,009	14,993,414	15,807,911	6,209,540	5,096,358
Received	108,601	83,535	140,717	148,248	310,240	296,975	323,209	328,239	401,252
Latenced	109	84	141	148	310	297	323	328	402
Mean	12,540	91,818	83,287	77,128	50,551	50,482	48,940	18,931	12,677

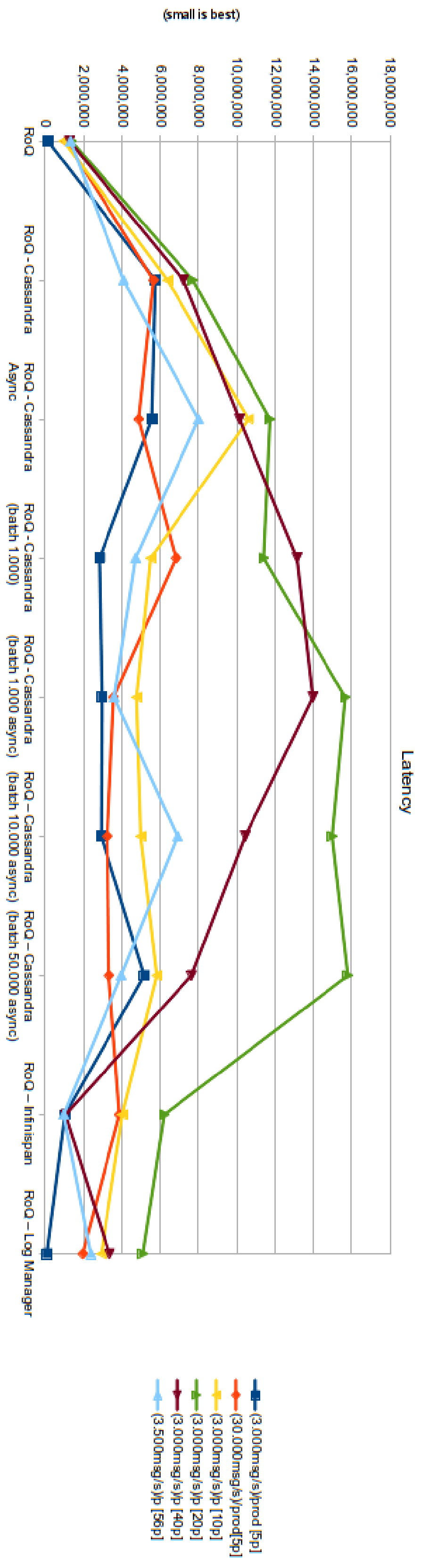
(3.000msg/s)/prod; 10prod*3.000 = 30.000 msg/s; 20*30.000 = 600.000 bytes/sec; 60*600.000 = 36.000.000
(3.000msg/s)/prod; 20prod*3.000 = 60.000 msg/s; 20*60.000 = 1.200.000 bytes/sec; 60*1.200.000 = 72.000.000

	RoQ	RoQ - Cassandra	RoQ - Cassandra Async	RoQ - Cassandra (batch 1.000)	RoQ - Cassandra (batch 1.000 async)	RoQ – Cassandra (batch 10.000 async)	RoQ – Cassandra (batch 50.000 async)	RoQ – Infinispan	RoQ – Log Manager
(3.000msg/s)/p [40p]									
Throughput	75,655,188	2,321,900	3,761,180	8,260,000	7,434,300	6,957,240	4,136,360	29,021,664	60,474,351
CPU	29	6	7	7	11	15	10	17	14
Memory (Mb)	12	124	106	205	303	218	365	231	98
Latency	1,270,500	7,222,202	10,139,369	13,146,743	13,968,696	10,434,934	7,631,586	1,060,242	3,350,463
Received	55,682	77,831	108,438	301,369	269,130	231,169	101,347	74,999	170,427
Latenced	56	78	108	302	269	231	102	74	170
Mean	22,687	92,592	93,883	43,532	51,928	45,172	74,819	14,327	19,708
(3.500msg/s)/p [56p]									
Throughput	87,174,066	2,734,980	2,618,960	7,632,140	6,275,040	6,628,260	5,679,640	28,948,143	71,815,905
CPU	25	13	20	19	19	23	20	19	17
Memory (Mb)	13	123	116	163	171	297	320	142	139
Latency	1,312,587	4,090,682	8,003,612	4,718,198	3,600,949	6,908,226	3,969,336	963,964	2,373,614
Received	46,238	77,024	86,686	94,812	75,428	81,613	87,737	71,011	64,856
Latenced	47	77	87	95	76	81	88	71	65
Mean	27,927	53,125	91,995	49,665	47,380	85,286	45,106	13,096	36,517

(3.000msg/s)/prod; 40prod*3.000 = 120.000 msg/sec; 20*120.000 = 2.400.000bytes/sec; 60*2.400.000 = 144.000.000
(3.500msg/s)/prod; 56prod*3.500 = 196.000 msg.sec; 20*196.000 = 3.920.000bytes/sec; 60*3.920.000 = 235.200.000

The following two graphs resume the results data from the previous table.





Appendix B

This appendix contains instruction for launching RoQ with Message Logging System.

The source code of RoQ with Message Logging System can be fond on the CD-ROM provided with this thesis. The classes of MLS are located in :

RoQ-master/roq-core/src/main/java/org/roqmessaging/{log, log/reliability, log/storage}

Requirements for running RoQ with MLS:

- Java 1.7
- ZMQ 3.2.3
- JZMQ 2.2.0

To lunch the RoQ with MLS follow the next steps:

```
go to /roq/bin and launch:
./startGCM.sh
```

Code 1: RoQ GCM launcher.

```
go to /roq/bin and launch:
./startHost.sh
```

Code 2: RoQ Host launcher.

For testing:

```
java -Djava.library.path=/usr/local/lib -Dlog4j.configuration="file:roq/
    config/log4j.properties" -cp roq/lib/roq-management-1.0-SNAPSHOT-jar-
    with-dependencies.jar org.roqmessaging.management.launcher.
    QueueManagementLauncher <addess of the GCM> add myqueue
```

Code 3: RoQ Queue launcher.

```
java -Djava.library.path=/usr/local/lib -Dlog4j.configuration="file:roq/
    config/log4j.properties" -cp roq/lib/roq-simulation-1.0-SNAPSHOT-jar-
    with-dependencies.jar org.roqmessaging.loaders.launcher.
    TestLoaderLauncher test.txt myqueue
```

Code 4: RoQ Test launcher.