High availability for RoQ core components

Master's thesis in collaboration with EURA NOVA for the graduation of Master in Computer Science option Software engineering
by Benjamin Van Melle

Promoter: Peter Van Roy
Copromoter: Alexandre D'Erman
Reader: Richard Gil Martinez

Université Catholique de Louvain-la-Neuve
Belgium
Academic year 2014 - 2015

Contents

Acknowledgements

All my acknowledgements to Alexandre D'Erman for his advices about my work during all the academic year. I also want to thank Sabri Skhiri who introduced me to the master's theses proposed by EURA NOVA, who helped me to choose this interesting subject and who critiqued my solutions. I'm grateful to Peter Van Roy for his comments about my work and his very interesting courses about computer languages and distributed systems. Finally, all my acknowledgements to EURA NOVA's employees for their host during my internship.

Part I

Introduction

Chapter 1

Context and contribution

This chapter is an introduction to this master thesis. We introduce some concepts which help understand this thesis and we describe the structure of the document and our contribution.

First, we introduce the concept of distributed system. Second, we present two common architectures to design such systems. Third, we present the CAP theorem which introduces an important limit of such systems. Fourth, we present the publish/subscribe communication paradigm. Fifth, we provide a brief introduction to RoQ. Sixth, we introduce the concept of high availability. Finally, we explain the contribution of this master thesis and the structure of this document.

1.1 Distributed systems in a nutshell

"A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility." [1] These autonomous machines are sometimes called *nodes* and the group of nodes which compose the system is called the *cluster*.

Distributed systems have many advantages but also some drawbacks. The most important advantages are the following:

- **Horizontal scalability**: In a single-node system the load that can be handled is limited by the host resources (e.g. CPU, memory, etc.). They are only *vertically scalable*, meaning that the only way to improve

the system's performance is to upgrade the physical components of the host machine. Once the best components are used there is no other way to improve performance. Distributed systems tackle this problem by allowing the system's performance to be improved by increasing the number of nodes.

- **Elasticity**: Some distributed systems are elastic. That means that they are able to scale up - by increasing the number of nodes - or scale down - by decreasing the number of nodes - the system automatically. The decision to scale (up or down) is taken by an algorithm which monitors the resource usage with respect to a policy. The policy indicates the *Service Level Agreement* (SLA) that the service must satisfy (i.e. its performance).

- **Fault tolerance**: A distributed system which is fault tolerant continues to provide its service when its nodes or processes crash. To deal with this situation, the system must implement a failure detector in order to detect crashes, and a recovery process to recover the system once a crash has been detected. A system which is completely fault tolerant has no single point of failure, meaning that a node or process crash cannot make the system unavailable.

The most important drawbacks are the following:

- **Network partitions can occur**: The nodes are coordinated by exchanging messages over a network. A distributed system must be able to face a situation in which the nodes are split in two or more subsets. Nodes can communicate within these subsets, but they can't with nodes belonging to others subsets.

- **Attacks are possible**: Since the nodes exchange critical messages over a network, many attacks become possible. Malicious people could send messages to nodes in order to attack them. If the system is not safe, the attacker could steal data, put the system in a bad state, etc.

- **Distributed systems are hard to validate**: It is more difficult to test a distributed system thoroughly in order to prove its validity. Because many situations can occurs (network partition, node crash, etc.), we have more scenarios to test compared to single-node systems.

1.2 Common architectures

During the next chapters of this master thesis, we will study distributed systems which are based on the following architectures: Master/Slave and Peer To Peer. This section provides a high-level overview of them.

1.2.1 Master/Slave architecture

This is a very common architecture in which we have two types of nodes: the *masters* and the *slaves*. A master is different from a slave because they run a different process (see figure 1.1).

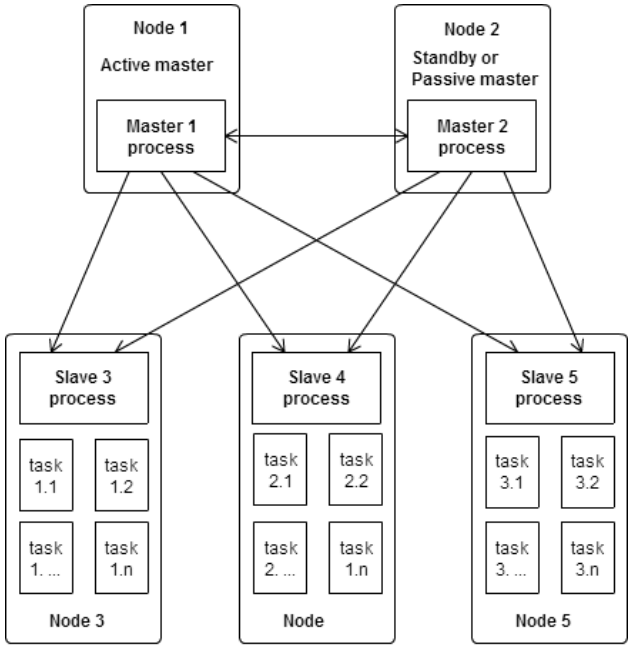

Figure 1.1: The master slave architecture

Slaves

The slaves are the nodes which actually provide the service. The processes they run depend of the service provided by the system. For instance, in HBase, each slave's process handles a part of the database, in Storm they handle a part of the topology, etc.

In this type of architecture, the slave is the unit of scalability. We increase (resp. reduce) the number of slaves to scale up (resp. scale down) the cluster.

Each slave is connected to the master and receives messages from it. These messages can indicate to the processes that the slave must start or stop, etc.

Masters

Generally, the masters are the nodes responsible for handling the cluster metadata. This data contains information such as the cluster's topology, the service's state, the processes which run on the slaves, etc.

Thanks to this information, the master is able to perform the following operations (non exhaustive list):

- Start or stop processes on the slaves.
- Monitor slaves in order to detect and recover them if they crash.
- Publish the cluster topology to the clients.
- Start or stop new slaves.
- Balancing the tasks between the slaves in order to optimize resource usage.

The master is often replicated on different nodes in order to be fault tolerant. In this case we distinguish two types of masters, the active master and the backup masters. The active master is the one which is responsible for performing the operations (such as handling clients' requests). The backups stay in standby and perform no operations. They are automatically activated if the active master has crashed in order to avoid single point of failure.

1.2.2 Peer to peer architecture

In this architecture all the nodes are equal, meaning that each node has the same components. In a peer to peer architecture, each node consumes resources from the others nodes and provides resources to them (they are both client and server). These resources can be files, memory, computation time, etc. The goal of this architecture is to use efficiently the resources available on the different peers.

A key point in a peer to peer architecture is the resource discovery [2]. Since the peers share resources, they must be connected together in order to form an overlay network (i.e. a network which is defined on top of another network[3]) on which the peers can exchange messages. We distinguish three types of such networks:

- **Unstructured network**: The overlay network is unstructured, meaning that each peer establishes connections with a random set of peers. A drawback of this approach is that if a resource is rare (e.g. a file available on only one peer), the peer which want to acquire this resource must flood the network (thus wasting resources). An advantage of this approach is that the network is resilient to node crashes. For instance, the peer to peer service Kazaa follows this architecture.

- **Structured network**: The overlay network is structured, meaning that each pear establishes a connection with an other peer according to an algorithm which defines a structured topology. An advantage of this approach is that a resource can be found by an algorithm efficiently. In this way, even the resource is rare, we don't need to flood the network. A drawback is that the network is less resilient to nodes crash. In the state of the art, we will present Cassandra, a distributed database which follows this architecture.

- **Hybrid model**: In this model, a central server maintains metadata about the peers such as the resources provided by each of them. In this way, before establishing a connection with another peer, the peer sends a request to the central server to get the resource location. For instance, the peer to peer service Napster followed this architecture.

1.3 CAP theorem

Distributed applications must deal with the results of an important theorem called CAP [4]. "The CAP theorem always applies for all distributed systems, at all levels of abstractions and at all sizes" [5]. Actually, the CAP theorem states the tradeoff between Availability, Consistency and Partition tolerance.

- Availability: "Every request eventually returns a result"[5].

- Consistency: "All system's operations are atomic (totally ordered)"[5].

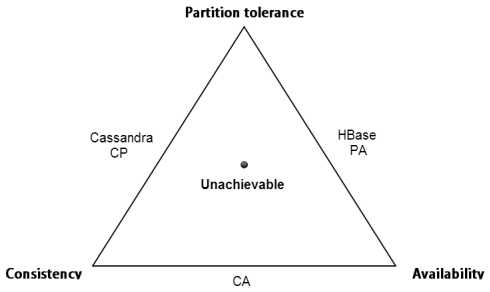

Figure 1.2: The CAP theorem

- Partition tolerance: "Any message may be lost"[5].

The theorem concludes that it's only possible to pick two of these three properties at the same time (figure 1.2). AP (availability-partition tolerance) means that the system stays available when a partition occurs. CP (consistence-partition tolerance) means that the system remains consistent when a partition occurs. Since network partitioning is an event which will eventually happen in a distributed system, system designers must choose between consistency and availability in such a situation.

Availability or consistency is not a binary choice. At a more fine grained level, system engineers can make different choices depending on the functionalities. For instance, a functionality (e.g. a read from a database) stays available, while another functionality (e.g. a write on a database) is not available. Thus, when a partitions occurs only some features are unavailable. Many choices resulting from the CAP theorem will appear during this thesis.

1.4 The Publish Subscribe paradigm

Publish-subscribe[6, 7] is a communication paradigm which connects two types of clients, called publishers and subscribers. They are connected via a common bus called messaging service. The publishers send messages to the messaging service which is responsible for delivering them to the subscribers. Subscribers have the ability to receive only the messages that match their interests. These interests may be registered in different ways (channel-based, topic-based, content-based or type-based) which offer different degrees of expressiveness and performance. The following are two opposite distribution methods:

- **Topic-based**: This mode of distribution associates each message with a specific keyword called topic. The subscribers register their interest for topics by contacting the messaging service. When a message for a topic is published, the service delivers them to all the subscribers that have subscribed for this topic.

- **Content-based**: The subscribers register their interest for specific messages via a filter, defined thanks to a subscription language. Such a language allows building constraints in order to filter messages. These constraints are based on logical operators and form complex subscription patterns.

The most interesting one for the current work is the topic-based distribution. It is the most efficient (but also less expressive) way to distribute messages.

1.4.1 Decoupling

The Publish/Subscribe communication paradigm is very interesting due to the low level of coupling that it provides between subscribers and publishers. A low level of coupling is a very desirable property in the context of distributed systems. By nature a distributed systems' component must be decoupled as much as possible in order to limit the required coordination mechanisms and provide better scalability and parallelization. The provided decoupling can be divided in three dimensions:

- **Time decoupling**: publishers and subscribers don't have to be available at the same time. For instance: the subscriber can be disconnected while the publisher is sending a message. In the same way the subscriber might receive a message while the message's publisher is disconnected.

- **Synchronisation decoupling**: The act of publishing or receiving messages is asynchronous.

- **Space decoupling**: publishers and subscribers do not have to know each other, publishers just have to provide messages to the messaging service which routes them to the subscribers.

Each publish/subscribe system has its specific position on these three dimensions. Generally, publish/subscribe systems are well decoupled along

space and synchronisation dimensions. Concerning time decoupling, each system has its own specificities. For instance, some systems don't allow a subscriber to retrieve messages that were published before it connected (RoQ). At the opposite end, some services have large backlogs which allow the subscribers to retrieve old messages (Kafka).

1.4.2 Quality of service (QoS)

An important aspect to consider regarding to publish/subscribe service is the quality of service (QoS), which varies from one system to another depending on the system's goal. The following list presents some characteristics to take into account when designing a publish/subscribe service.

- **Large backlog**: Messages are stored in the system to ensure that they remain available for a specific period. A large backlog allows a subscriber to get any message submitted during this period.
- **Message persistence**: The persistence ensures that messages are stored in the system as long as they have not been delivered to their connected subscribers. Thus the system must ensure that messages can survive to node crashes.
- **Delivery semantic**: Publish/subscribe services propose three delivery semantic: *at least once*, *at most once*, *exactly once*.
  - *At most once* means that each message is delivered at most one time (0 or one time). i.e. a message can be lost.
  - *At least once* means that each message is delivered at least one time. It is more reliable than *at most once*, but messages can be duplicated.
  - *Exactly once* means that each message is delivered exactly one time. This is the strongest property but also the hardest to achieve.
- **Message ordering**: Messages are delivered in a specific order to the subscribers. We can order the message by priority or in a FIFO (first in first out) order.

  - Priority: Each message has a priority level. The messaging service chooses the next message to deliver according to the priority of the buffered messages. Messages with the highest priority will be delivered first.
  - FIFO order: Messages are delivered in the order they are received by the messaging service.
  - Unordered: The service provides no guarantee about the order of delivery.

- **Transaction**: Messages can be grouped into atomic units (called transactions) which are either fully delivered or not at all. If a producer fails while sending messages from a transaction, the messaging service will not deliver anything.

1.5 High Availability

A highly available (HA) system minimises the planned and unplanned downtime.
Planned downtime refers to the time required to perform a maintenance operation (such as the deployment of an update), while the unplanned downtime refers to the time where the system is not available for an unexpected reason such as a network partition, a node failure, etc. We calculate the system's availability with the following formula:

$$Availability = uptime/totaltime = (totaltime - downtime)/totaltime \tag{1.1}$$

During a downtime period the service is considered unavailable (e.g. the system doesn't respond to user's requests). The availability can be expressed as a percentage of runtime, as calculated with the previous formula. It is common to consider the downtime per month, year, or week. The following grid show the relationship between downtime and availability.

Percentage	Per year	Per month	Per week
90% ("one nine")	36.5 days	72 hours	16.8 hours
99% ("two nines")	3.65 days	7.20 hours	1.68 hours
99.9% ("three nines")	8.76 hours	43.8 minutes	10.1 minutes
99.99% ("four nines")	52.56 minutes	4.32 minutes	1.01 minutes
99.9999% ("six nines")	31.5 seconds	2.59 seconds	0.605 second

Table 1.1: Relation between downtime and availability

An highly available system tries to reach the highest availability by reducing the downtime and the faults which can occur and by removing single points of failure.

1.6 RoQ

RoQ development began in 2011 at Eura Nova. It is under the Apache License v2 and is currently a work in progress. RoQ's implementation is based on EQS (Elastic Queue Service), a master/slave architecture which Eura Nova proposed in 2011 in order to provide a publish/subscribe service designed for the cloud. The main motivation for this work was to create an elastic architecture based on a scaling algorithm. Elasticity is the ability of a system to scale out - increase the number of brokers - or scale in - reduce the number of brokers - in order to adapt the system's capacity to the workload. In a publish/subscribe service, the workload is expressed in terms of throughput - number of messages that the system handles by unit of time - and of number of publishers and Subscribers. Elasticity leads to better resource usage in order to optimize infrastructure costs.

1.7 Contribution

Following this introduction to the problem, part II will focus on the state of the art while part III will describe the chosen solution, its implementation and its validation through testing. The final part is a conclusion about this master thesis and provides some directions for future works on RoQ.

We introduced in this chapter the basic concepts necessary to understand distributed systems and more exactly the publish/subscribe communication

paradigm. This way, we understood that making such systems highly available while providing some guarantees (i.e. delivery semantic, message persistence, etc.) is not a straightforward problem. This is why the next step is to study distributed systems and the way to make them fault tolerant.

The *state of the art* (part II) provides an analysis of six cutting edge technologies which solve common problems for different types of services. One of these problems is fault tolerance, a problem more related to the system's architecture than to the service that it provides. For this reason, the state of the art introduces technologies from different fields such as: distributed databases, stream processing and publish/subscribe systems. Each case study focuses on a description of the system's architecture and the solutions used to make it fault tolerant and highly available. The chapters are the following:

1. **ZooKeeper**: We present ZooKeeper, a service for coordinating distributed systems. First, we study the service and its features. Second, we study its architecture and the guarantees that it provides.

2. **Distributed databases**: We present two important distributed database systems based on two different architectures and which take different approaches to the CAP theorem. The first is Cassandra, a database system developed by Facebook and based on a peer to peer architecture. Cassandra has chosen to stay available rather than consistent when network partitions occur. The second is HBase, Hadoop's solution based on a master/slave architecture which prioritizes consistency. HBase is interesting in the context of this thesis because of its architecture which is very close to RoQ's.

3. **Stream processing**: This chapter presents Storm, a technology developed by Twitter which provides a framework to compute data streams in real time. Storm is widely used in production and proposes a mature technology. It has a master/slave architecture designed to be fault tolerant and to provide some guarantees such as at least once computation semantic.

4. **publish/subscribe**: This chapter is about RabbitMQ and Kafka, two of the most popular publish/subscribe systems which provide interesting guarantees and features. In the scope of this thesis, they are more interesting for the delivery semantic than for their high availability implementations because their architectures are very different from RoQ's.

At the end of the state of the art, we will have a good understanding of the master/slave architecture and how to make it fault tolerant. In addition, we will have a good introduction to some of the most popular cutting edge technologies. Finally, this chapter has an interesting approach regarding distributed systems and the problems encountered in this area of computer science.

The part on *High Availability for RoQ* (part III) is made of five chapters which provide an implementation of high availability for RoQ. The chapters are the following:

1. **RoQ**: We describe RoQ's architecture in details in order to understand how it works and what are its single points of failure.

2. **Abstracting solutions**: We formalise two of RoQ's problems (availability and delivery semantic) and we provide abstract solutions to solve them based on the state of the art.

3. **High level solution**: We propose a high level solution based on the abstract one from the previous chapter. At the end of this chapter, the solution for high availability is ready to be implemented.

4. **Implementation details**: We describe the implementation and some algorithms more in detail.

5. **Tests and edge cases**: We discuss about the tests that were developed to ensure the validity of our solution and the various edge cases that are not covered by it.

Part II

State of the art

Chapter 2

Coordination with ZooKeeper

2.1 Introduction

ZooKeeper [8] is a service for coordinating distributed systems. It was released under the Apache license in 2012. It proposes simple kernel but powerful *primitives* that can be used for building coordination mechanisms (called *recipes*) such as leader election, service discovery, simple locks, read-write locks, consensus, etc. In this part of the thesis, we will present the service, the system's architecture, an example of recipe built thanks to these simple primitives. Finally, we will introduce Curator [9], a client library created by Netflix which provides reliable recipes.

To provide these features, ZooKeeper relies to a distributed and consistent metadata store on which the clients can bring modification concurrently and can receive notification when a modification on the store has been performed.

2.2 Service overview

2.2.1 Data model

In ZooKeeper, data is stored in a hierarchical name space which looks like a traditional file system. Figure 2.1 shows a ZooKeeper *data tree* with six nodes. At the top we have the root node (/) with its two children (/app1 and /app2). The nodes are organized and referenced like files in the UNIX file system. A Zookeeper's node is called a *znode* and contains data. The data stored in a node is bounded by default to 1MB to avoid using ZooKeeper as

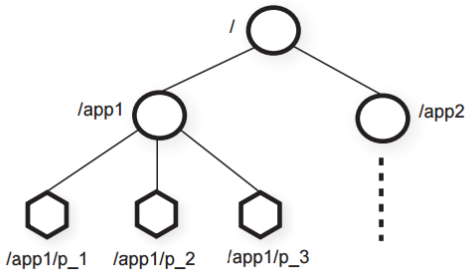

Figure 2.1: ZooKeeper hierarchical name space, taken from [8]

a large data store. Znodes are designed to store only few KB of data (e.g. instance a configuration parameter, a service state, etc.).

2.2.2 Session

To create, read, modify or delete a znode in the service, a client must establish a *session* before it can send its requests.

A session in ZooKeeper is maintained for a configurable timeout. The client sends a periodic heartbeat to the service. When the service has not received this one for a period longer than the timeout. The session is considered by ZooKeeper as closed. The session is used to provide some features.

2.2.3 znodes features

Watchers

The services allows the client to register a *watcher*. When it watches a znode, ZooKeeper sends it a notification when a modification has occurred on this one (e.g. the znode's data has changed or the node has been removed). Once that a notification has been sent, the watcher is unregistered and the client must recreate it. To watch a znode, it must send a simple "getData" or "exist" request with the watcher parameter set to true.

ephemeral and regular znodes

There are two type of znodes:

- **Regular znodes** are explicitly created and removed by clients.

- **Ephemeral znodes** are created by clients and can then be removed in two ways: either explicitly by the clients or by the service itself if the session with the client who made the znode has been lost.

Sequential znodes

When it create a new znode, the client can specify the *sequential flag*. When this flag is set to true (false by default), the systems appends a monotonically increased value to the path name. For instance, if the client create the sequential znode /test, the resulting path will be /test-0, if it creates this znode again its path will be /test-1.

2.2.4 Example

Now that the service has been introduced, we can describe a common recipe. The example described here is a *leader election*. Leader election provides the ability to elect one and only one client as the leader and to automatically elect a new one if the current leader fails.

To implement leader election, we first create an unique base path dedicated for this recipe (e.g. */leaderElection*). We distinguish two situations, the election and the leader crash detection.

To elect a leader, all the clients register first a ephemeral and sequential znode under the base path. These znodes must have the same name (e.g. */leaderElection/node-*). Because all these nodes are configured to be sequential, each of them will have an unique name thanks to the value appended to the path. In this way, we have a set of znodes {*/leaderElection/node-j*; j >= 0 and where each node-j corresponds to a client}. The leader is the client which has the smallest j in this set.

To detect failure and elect a new leader efficiently, each client is watching its previous node. i.e. if the client has the znode */leaderElection/node-j*, it is watching the znode */leaderElection/node-i* where i < j and there exists no k such as k < j and k > i. If the znode that it is watching is removed, the client become the new leader.

We will see in the next chapters of the state of the art that this recipe is commonly used in distributed systems.

2.3 Overview architecture

2.3.1 Cluster

A cluster in ZooKeeper is made of several *servers* (a server is equivalent to a node, but to avoid confusion with znodes, we talk about servers in this section). Each of them has the same component. We distinguish two type of servers, the *leader* and its *followers*. We also distinguish two type of requests:

- **Write request**: a request which modifies the database state (e.g. write, update, etc.).
- **Read request**: a request which not modifies the database state (e.g. read, exist, etc.).

Although both the leader and its followers can process read requests, only the leader is able to process the write requests. Hence, when a follower receives a write request, it routes this one to the leader. For this reason, the servers in ZooKeeper are connected to each other. The leader to its followers in order to broadcast the modifications, and the followers to their leader in order to rout write requests.

A client can establish a connection with any servers, however, it is better for them to have the ZooKeeper' servers list in order to be able to send request to another server if the first one has crashed (fault tolerance).

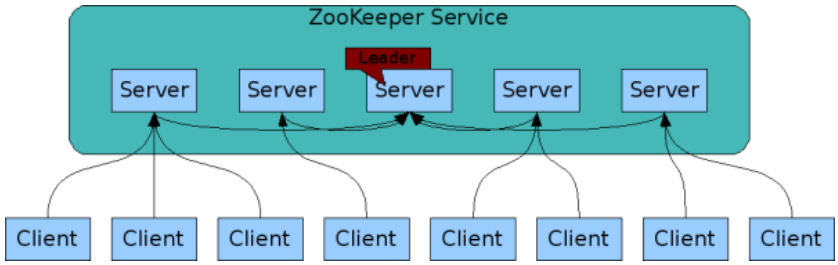

Figure 2.2: ZooKeeper overview, taken from [10]

In next sections, we will describe the servers' components, their guarantees and the position of ZooKeeper regarding to the CAP theorem.

2.3.2 Components

Figure 2.3 shows the servers' components: the *Request Processor*, the *Atomic Broadcast* and the *Replicated Database* (the RequestProcessor is exclusively used by the leader).

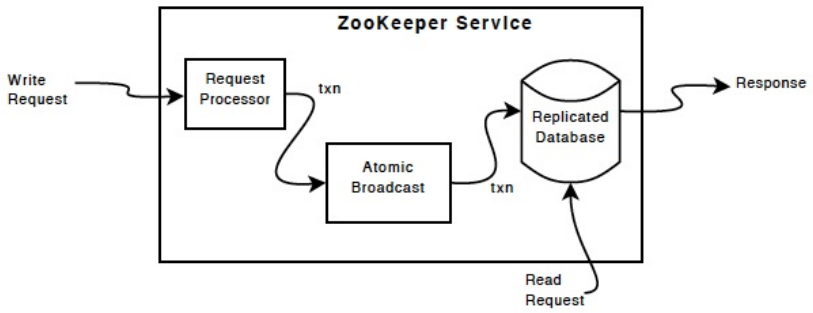

Figure 2.3: ZooKeeper components, taken from [8]

Request Processor

When the leader receives a write request it first transforming this one to an idempotent transaction which captures the next state. For instance, a setData performed by the client will be transformed to setDataTXN transaction. This one contains the information to perform the idempotent setData. Note that if the client try to perform a write request on a follower, this one is automatically routed to the leader.

Atomic Broadcast

The Atomic Broadcast is the component which ensures the coordination among the servers. When the leader performs a modification on the data tree, the TXN transaction resulting from the Request Processor is broadcast to all the followers. The broadcast protocol used by ZooKeeper relies to Zab [11] which is a "crash recovery broadcast algorithm" according to [11]. Zab was designed specially for ZooKeeper but we are confident about other use cases. The specificity of Zab is that it ensures that if it broadcast a transaction, all its dependent transactions are first delivered. In this way, we can ensure that the replicas (located on the followers) are eventually in the same state as the leader. Finally, Zab performs the leader election for the service.

Replicated Database

Each server (replica) has its own copy of the database which contains the data tree. The replica takes a snapshot of this one periodically. This snapshot is used to accelerate the crash recovery process in order to avoid to reapply all the delivered transactions. The client allows to perform read requests on any server, while write requests are only performed at the leader. Therefore, a read request does not always fetch an up-to-date result, but presents the advantage to accelerate the read throughput. If the client must get the latest version of the znode, it must perform a *sync* first. This operation ensures that all the modifications applied to the data tree before the sync call have been applied to the replica.

2.3.3 Guarantees

When performing lot of requests concurrently in ZooKeeper, the order becomes important. The system provide two order guarantees:

- **Linearisable writes**: The clients send requests concurrently to ZooKeeper. These ones are applied on the same order to all the ZooKeeper nodes. In this way, the consecutive states resulting from these requests are consistent. i.e. If a client receives 0 and then 1. That means that 0 was written before 1.
- **FIFO client order**: The client's requests are executed in the same order as they were sent.

ZooKeeper is a data storage which provides consistent data to the clients of a distributed system. With these guarantees, ZooKeeper allows to build complex and distributed algorithms reliably such as leader election, user-groups, consensus, failure detection, etc.

CAP theorem

We saw previously that ZooKeeper distinguishes two types of request, the writes and the reads. In addition we saw an additional request called sync. The position of the service regarding to the CAP theorem depends of the request's type. In the presence of a partition between a follower and its leader, the sync and the writes requests sent on the follower are unavailable, because they must be routed by the follower to the master. Therefore, the write

requests tradeoff consistency over availability. At the opposite a simple read request is always available, the result received by the client will reflect the current state of replica. We can qualify ZooKeeper as eventually consistent because the replicas' state will converge to a consistent state if we don't perform operations on the cluster for some time.

2.4 Curator

Curator [9] is an higher level client for ZooKeeper developed by Netflix. It provides ready to use recipes which allow the developer to avoid to reinvent them. We saw previously how to build a double barrier with ZooKeeper but not how to do it reliably. Actually a client can encounter some problems when it tries to perform a request on ZooKeeper, for instance it cannot establish a connection with the server, etc. Curator takes care of these details for us, it implements many common recipes reliably. A non exhaustive list of curator recipes is: *Leader Election, Shared Reentrant Lock, Shared Semaphore, Barrier, Double Barrier, Shared Counter, Node Cache, Tree Cache, etc.*

2.5 Conclusion

We presented ZooKeeper, a specialized storage solution for coordinating a set of clients thanks to its features. We saw that ZooKeeper is not appropriate to store large pieces of data because of its complex mechanisms to ensure order guarantees and state consistency. For this reason, it must not be confused with the distributed databases presented in next chapter. The use cases are very different. First because ZooKeeper is not designed to be write intensive, second because it doesn't provide linear scalability. Common Zookeeper's cluster is composed of three or five nodes. These values correspond to the most efficient ones.

Chapter 3

Distributed Databases

3.1 Introduction

3.1.1 Technology overview

Distributed databases are today very common in the landscape of the computer science. There had been introduced in response to the lacks of the traditional relational databases which run on a single machine. The major issue with them is their low scalability.

The distributed databases face an important challenge. It consists to be able to store a very important quantity of data and ensure quick access time. In this way, developers can make new kinds of application which work with a larger amount of data called *Bigdata*. Bigdata designates data sets which are too important to be managed by the traditional database management systems. The application domains of these new applications are numerous (Search engines, astronomic computations, social networks and so on).

3.1.2 ACID vs. BASE

In the landscape of the databases we distinguish two sets of guarantees (ACID and BASE). Traditionally relational database management systems (RDBMS) were focus on four guarantees called *ACID*. They aim to ensure the consistency of the data during database lifetime. ACID properties relate to the concept of transaction. A *transaction* is a group of operations which modify the system state in order to reach a new one. Now that a transaction is defined, we can present the ACID guarantees:

- **Atomicity (A)**: All the instructions inside a transaction are executed. If something fails during the transaction execution, the transaction is rolled back to the initial state (the state before the transaction).
- **Consistency (C)**: Before and after the transaction execution, the system is consistent. i.e. The state respects the rules defined by the system. Its important to distinguish this concept from the consistency of the CAP theorem.
- **Isolation (I)**: The operations of the transaction have no effect on the intermediate state of the others transactions (absence of race conditions).
- **Durability (D)**: Once a transaction finishes, the new state is definitive. i.e. the new system's state survives to crash (e.g. system shut down, failure, etc.).

ACID Guarantees are very desirable if we want a robust system on which the operations are safe. Some distributed systems choose to relax these guarantees in order to propose an alternative way which can deal with the drawbacks of a distributed environment such as the network partitions, the nodes failure, etc. These guarantees are called *BASE*:

- **Basically Available (BA)**: The system is available regarding to the CAP theorem. If a node fails or a network failure happens, the system remains available and requests continue to be processed on the available nodes.
- **Soft state (S)**: The state of the system is not freeze when no transactions are in progress. The nodes continuously share data and modify their state.
- **Eventual consistency (E)**: Means that the state of the nodes eventually converge to a consistent state.

These properties fit well with distributed systems because they allow the system to handle requests when a node crashes or a network partition occurs. For instance, BASE guarantees allow the database to provide old data to the client. That could be acceptable for a search engine but not for a banking system.

3.2 Cassandra

3.2.1 Origin of Cassandra

Cassandra [12] is a distributed database designed by Facebook. It has been developed to solve the problem of storing the users messages in the best way to provide efficient search. This technology is a peer to peer distributed system which aim to be horizontally scalable and highly available. In addition Cassandra is able to replicate nodes across multiple datacenter in order to provide low latency and better user experience for people around the world. Even though Cassandra was developed to respond to a specific problem, it can be a solution for multiple systems.

3.2.2 Data model

Cassandra is a hybrid between column-oriented and key-value database.

Cassandra rows are referenced by *keys* (see figure 3.1). Together these keys form the *keyspace* (the set of all the keys). We distinguish three different types of columns.

- The first is the columns family. these columns provide meta informations about the columns that compose the rows (for instance column name and type). Actually, each row might have a different set of columns.
- The second is the *column*. These columns are tuples that contains a name (green on the figure), a value (optional) and a timestamp. The timestamp indicates the most recent value for a given column.
- The last kind of column is the special column. There exists three types of special column:
  - **super column**: This is also a kind of column container that adds a level to the structure. The super column contains a set of columns.
  - **counter column**: Maintains a counter that can be incremented (or decremented). This structure is interesting because this counter is consistent across the replicas.

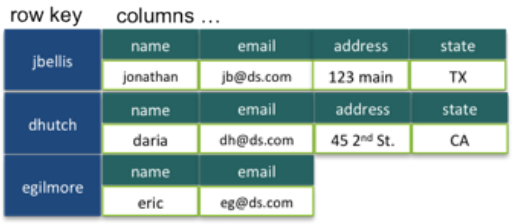

Figure 3.1: A static column family, taken from [12]

– **expiring column**: This column maintains an expiration date (TTL) after which the column is removed.

3.2.3 Cluster topology

All nodes in Cassandra are equal, meaning that there are no master nodes responsible for specific functions.

The protocol used by the nodes to get information about their location is called *Gossip*. In this protocol, nodes exchange messages regularly to maintain correct knowledge about the system topology. In Cassandra the shape of the topology is a *ring* (see figure 3.2). In Cassandra nodes exchange gossip messages each second. A gossip message contains information about a node itself and about the other nodes that it knows. The information contained in the message holds a version, so that during the merging of data, the node keeps only the most recent version of the information.

3.2.4 Partitioning

This section discusses about the fair repartition of the keys among the nodes. The keys about which we talk here are the keys from the keyspace (see data model section).

A Cassandra table is deployed on a cluster on which the nodes take a position on the ring (see figure 3.2). Each node and each value is located according to a token evaluated by consistent hashing (described bellow). A value identified by token k is stored on the first node which follows k (we call this node the successor of k). One of the particularity of the consistent hashing is to ensure that the tokens are well spread among the nodes, hence each node tend to have the same quantity of tokens under its responsibility.

This model is called the chord ring [13]. One of the best advantage of chord is to minimise the modifications due to the introduction (or removal) of a node into the system. When the node leaves the successor of the node is now responsible of its tokens. When a node enters the successor gives a copy of the data that becomes under the responsibility of the new node.

Partitioner is the function used in consistent hashing to derive the token which will be used to determine the position of the keys on the ring. A good partitioner ensures a fair repartition of the keys among the nodes (to respect consistent hashing). There exist three kinds of partitioner in Cassandra:

- **RandomPartitioner** creates a token based on the MD5 hash value of the row key to distribute data across the nodes. The range of hash values is between 0 and $2^{127} - 1$.
- **Murmur3Partitioner** (default) is a refinement of the RandomPartitioner which provides faster hashing and better performance. The range of hash values is between -2^{63} and $2^{63} - 1$.

A problem to consider is the heterogeneity of the nodes capacities. The numbers of keys assigned to a node is defined by the distance between the node and it predecessor on the ring. The distance is the range of keys under the responsability of the node. A way to adapt the key range of a node proportionally to the hardware capacity of the node is to use virtual nodes. A real node contains one to many virtual nodes, hence by adding (resp. removing) a virtual node you can increase (resp. reduce) the number of keys handled by the real node. Because consistent hashing ensures that each virtual node handles approximately the same quantity of keys, we can ensure that a node with more vnode will handle more keys.

3.2.5 Replication

Cassandra replicate the nodes to achieve durability (i.e. persistence in the case of node failure) and high availability [14]. The number of replicas of each node depends on the replication factor defined in the configuration file. A replication factor of one (resp. three) means that there is a single (resp. three) copy of each row in the system. Data is first assigned to a responsible node (see previous section). This node is responsible of the replications of the data within its range. Cassandra provides two strategies to achieve replication.

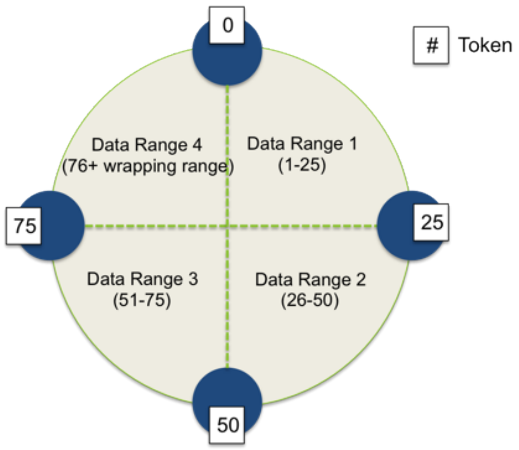

Figure 3.2: A Cassandra ring with four nodes, taken from [12]

- **SimpleStrategy**: The first node is placed according to the partitioner. Replicas are placed on the next nodes clockwise. This strategy is datacenter and rack unaware.
- **NetworkTopologyStrategy**: this strategy allows to specify how many copies of each node are placed in each datacenter. In addition the replicas are placed on different racks (improving fault tolerance due to rack crash). In each datacenter, first node is placed according to the partitioner. The replicas are on different racks and are placed by walking the ring clockwise until reaching the first node in another rack. There are two common configurations.
  - Two replicas by datacenter, that allows to tolerate the failure of one node if we read/write at a consistency level one (see next section).
  - Three replicas by datacenter, that allows to read/write at a consistency level of local quorum. In this case, the system supports failure of one node. Or at a read/write consistency of one. In this case, the system suppors the failure of two nodes.

3.2.6 Consistency level

Cassandra provide tunable read and write consistency. That allows the user to make it own choice between consistency and availability (see CAP theorem). We distinguish six level of read consistency and eight of write consistency.

The different levels of write consistency are the following:

- **Any** The write could be done only in the log of the hinted handoff (see last section).
- **One** The write is performed on the log of one replica and its memory.
- **Two** The write is performed on the logs of two replicas and their memory.
- **Three** The write is performed on the logs of three replicas and their memory.
- **Quorum** The write is performed on the logs of a majority of the replicas and their memory.
- **Local quorum** The write is performed on the logs of a majority of the replicas (in the same data center as the coordinator node) and their memory.
- **Each quorum** The write is performed on the logs of a majority of the replicas in each data center and their memory.
- **All** The write is performed on the logs of all the replicas and their memory.

The different levels of read consistency are the following:

- **One** The read is performed on one node.
- **Two** The read is performed on two nodes.
- **Three** The read is performed on three nodes.
- **Quorum** The read is performed on a majority of the replicas.

- **Local quorum** The read is performed on one a majority of the replicas (in the same data center as the coordinator node).
- **Each quorum** The read is performed on a majority of the replicas in each data center.
- **All** The read is performed on all the nodes.

By combining the replication factor and the read/write consistency level, it's possible to achieve different level of consistency. Basically it W (resp. R) designates the number of replicas on which the write (resp. read) is performed and N the replication factor. We could achieve (see background to understand the different levels of consistency):
Timeline consistency if R + W > N,
Eventual consistency if R + W <= N

3.2.7 Snitching

A snitch provides informations about the location of a node (rack and datacenter). Thanks to these informations about the topology of the cluster, Cassandra can distribute the replicas of the nodes in the most efficient way. For example by avoiding to replicate a node in the same rack or by replicating nodes across datacenters.

3.2.8 Failure detection

The failure detection mechanism of Cassandra uses the gossip protocol and the Φ accrual detection mechanism. With a Φ accrual failure detector [15], each node maintains a value called Φ for every other node. This value represents the level of suspicion of failure of the other node. A node is suspected if it didn't send a gossip message for a too long period. Higher is the threshold, longer is the period. Accrual detection mechanism adapts its threshold dynamically to take into account the possible heartbeats delay due to network issues such as a slow down, a high workload etc.

- **Strong completeness**: "There is a time after which every process that crashes is permanently suspected by all correct processes." [15]
- **Eventual strong accuracy**: "There is a time after which correct processes are not suspected by any correct process." [15]

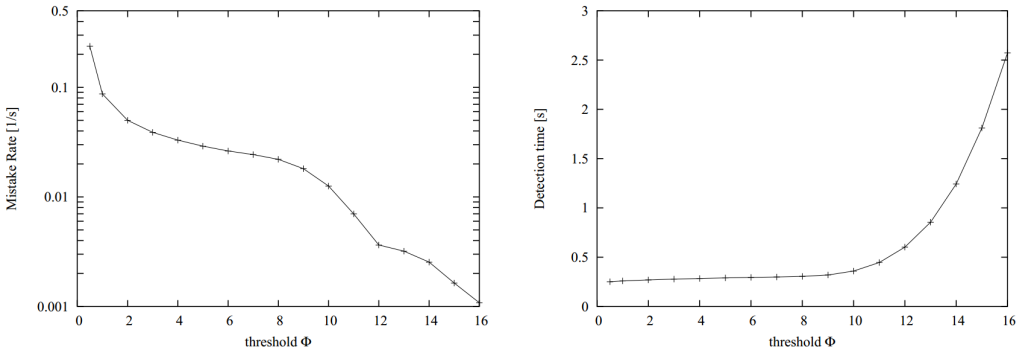

Figure 3.3: time to detect failure vs. mistake rate according to the threshold, taken from [15]

The system have to specify an initial threshold (see figure 3.3). A low threshold reduces the time to detect failures but increases the mistakes rate(i.e. the frequency of detection of a not crashed node) while high threshold increases the time to detect failures but reduces the mistakes rate.

3.2.9 Failure recovery

Cassandra introduces a mechanism called *hinted handoff*. It tackles the issues due to network failures or crashed nodes. The hinted handoff mechanism begins when a replica node doesn't respond to the write request sent by the coordinator into a bounded time. The hinted handoff mechanism consists to store at the coordinator the local changes (called *hints*) to perform at the not responding nodes. These local changes are stored for a limited time (3 hours by default) in the *hints table*. Once this time elapsed, the node is considered as definitively dead and the local changes are removed. When gossip indicate a node recovery, the coordinator sends the hints to update the recovered node.

3.3 Hadoop distributed file system

3.3.1 Introduction

Before introducing HBase, which is the next distributed database studied. We must introduce the hadoop distributed file system (hdfs) [16]. HDFS aims to propose a file system that performs standard operations on files (open, close, mkdir etc.). It is distributed in order to handle large amount of files. HDFS is a highly fault tolerant distributed file system that runs on low-cost machines. The system has the following main requirements:

- **Hardware failure**: In this kind of system hardware fault is very common. HDFS must detect faults quickly and recover from them.
- **Large data sets**: The system must support very large files (up to giga or tera bytes).
- **Simple coherency model**: The system is based on a write-once-read-many access model. This use case means that once the file is created and closed, It will not be opened again to edit data. This assumption (based on map-reduce use case) helps to improve data access throughput.

3.3.2 Architecture overview

Figure 3.4 shows an overview of the architecture of HDFS. It is based on a master/slave architecture, where the master node (called *NameNode*) manages the file system *name space* and regulates the clients files access. The slave nodes (called *DataNodes*) manage the storage of the data. The files are split in smaller blocks which are spread among the Data Nodes.

The NameNode provides operations in relation with the name space such as opening, closing and renaming files or directories. The NameNode is also able to determine the DataNode on which a block of a particular file resides.

The DataNodes are responsible for read and write requests. Therefore, they handle blocks creations, replications and deletions.

3.3.3 Replication

As we can see on the figure 3.4, the blocks are replicated for fault tolerance. The block size and replication factor for each file is set on the NameNode.

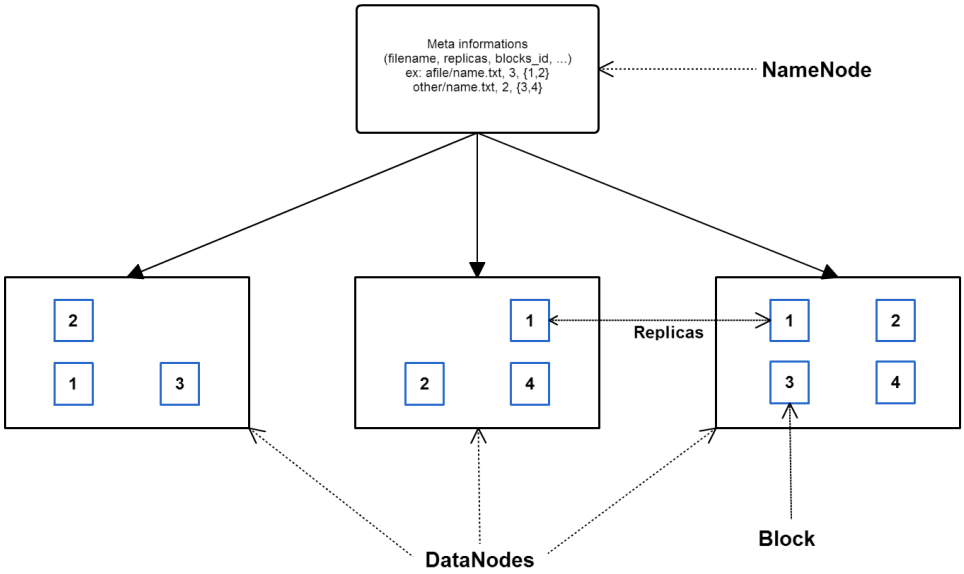

Figure 3.4: HDFS overall architecture, based on [16]

DataNodes, as long as they are alive, send regular heartbeats and block reports to the NameNode. A block report contains the list of all blocks of the DataNode. When a node doesn't send a heartbeat within a specified time, the node is considered dead and the NameNode runs a replication process that will replicate the block on another DataNode (to maintain replication factor).

3.3.4 Data persistence

The persistence of the meta information on the NameNode relies on a log mechanism which registers all the modifications made to files. The log is stored on a OS system file. The NameNode contains another file called the fsImage, this file contains all the meta informations (the name space) of the HDFS. To avoid corruption of these files, the system maintains multiple copies of them. At startup, the HDFS takes the last consistent fsImage and applies modifications from the log.

3.3.5 High availability

The NameNode is the single point of failure in this system. The availability of the system is impacted in two ways. In the case of an unplanned crash of the NameNode the HDFS becomes unavailable and a manual restart of the machine is necessary. In the case of a planned update of the NameNode software or hardware, the cluster is not available. In some use case of the HDFS, unavailability is not acceptable. It's why the high availability HDFS [17] has been developed. With this feature the single point of failure is avoided thanks to a replication of the NameNode and a set of deamons called "JournalNodes" (JNs).

One of the two NameNode is in active state while the other are in standby. Rather than register modifications brought to the NameSpace in the log file, modifications are registered in the JNs daemons (quorum based write). The standby node is constantly reading the log from the JNs to update its own NameSpace. In this ways, it stays synchronized with the active NameNode. At this point, the standby NameNode doesn't receive the informations about the blocs. To solve this issue, the DataNodes send heartbeat and blockreport to both. In this way the two NameNodes are totally synchronized. To elect an active node and detect the crash of one, the mechanism relies on the leader election feature of ZooKeeper.

3.4 HBase

3.4.1 Origin of HBase

Hbase development started in 2006 just after the publication of the BigTable paper. While BigTable runs on top of the Google File System (GFS), HBase runs on top of the Hadoop distributed file system (HDFS). The arguments supporting the development were the following:

1. It has been proven that BigTable is an efficient scalable architecture.
2. HBase is an opportunity to increase the tasks which can run on top of the HDFS.
3. It's a way to contribute to the growth of the hadoop ecosystem

The data model of HBase is globally the same as cassandra (column oriented).

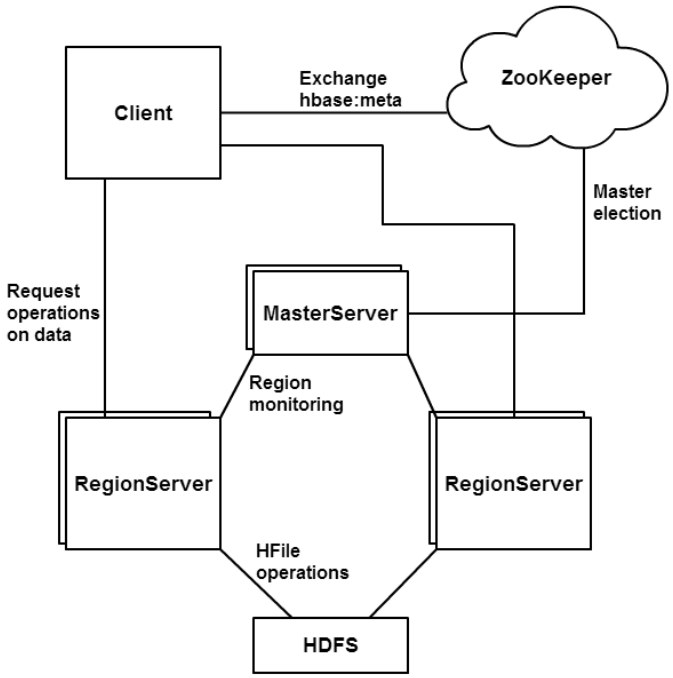

Figure 3.5: An overview of HBase architecture

3.4.2 Architecture overview

Compared to Cassandra, HBase apparently has a more complex architecture (see figure3.5). It comes from the fact that it is made of different nodes while Cassandra is made of node that are all equal (because it was a peer to peer architecture). There are two nodes specific to HBase (MasterServer and RegionServer) in addition to the ZooKeeper and HDFS nodes. The architecture of HBase aims to allow efficient random access and efficient read write on big data store (response to the issues of HDFS). The next paragraph describe the components of the figure 3.5.

MasterServer

The *MasterServer* is responsible for monitoring the RegionServers and for meta data of the tables. Usually it runs on the same server than the NameNode of the HDFS.

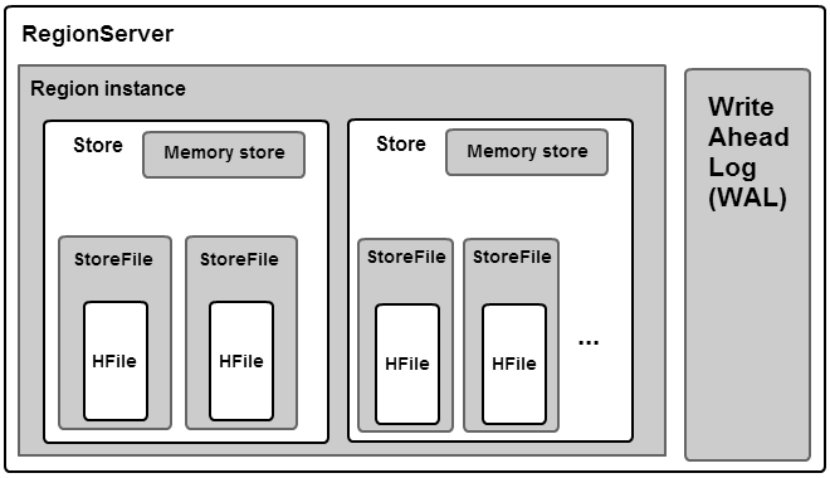

Figure 3.6: A region server architecture

- Its main operations consist of assigning, moving and unassigning regions to servers, add, remove and modify the columns of the column families ant create, remove, modify, enable and disable tables. The master is also responsible of the loadbalancing of the cluster.
- The load-balancing process is run every five minutes (by default) to equalize the number of regions by RegionServer.
- Finally, the MasterServer is responsible to set up the regions replicas (see "Timeline-consistent High Available Reads" section).

This server is stateless it's why it cans rely to the same mechanism as the NameNode (see HDFS section) to be highly available.

RegionServer

First, the *RegionServer* (see figure 3.6) commonly runs on the same server as a DataNode (of the HDFS) and provides of two types of operations.

- First, it provides data manipulation operations such as get, put, delete, next (scan feature), ...
- Second, it is responsible of maintenance tasks such as splitRegion (consist to split an existing region by reallocating the key range among two

nodes), compactRegion (consist to merge small files in a bigger one), ...

As we can see on figure 3.6, the regionServer is made of one region instance and one log file.

- A *region instance* maintains a store instance for each column family. Each *store instances* contains a memory store and zero to many store file instances. The *memory store* maintains the recent modifications brought to the store (which has not been flushed to the HDFS). When the memory store reaches its size limit, data is written into HFile. A *HFile* contains parts of the data composing a column family and is handled by the HDFS. The maxium fileSize is specified in the HBase configuration file. It contains actually four types of blocks.
  - **The trailer block** has pointers on the first index blocks and the file info block.
  - **The Index block** contains the offsets of the data blocks according to their key.
  - **The data block** contains the table data.
  - **The file info block** gives information about the compression type of the files the number of cells stored, the average size of keys, etc.
- Second, the *write ahead log file (WAL)* [18] is a file shared among all the store instances of the RegionServer. This file is written sequentially and maintains information about the modification of the stores which have not been registered in HFile. The entries of the WAL are called WALEdits, a WALEdit represents a transaction which can contains one or more operations. Thanks to the log, modifications brought to the store are persistent and survive to a node crash. In this way, the log ensures the durability of the data stored in the memory store.

ZooKeeper

HBase uses different services of the *Zookeeper* cluster. Zookeeper is used for coordination among nodes and for electing a master between the replicas. The starting point of a HBase request by a client is the contact with ZooKeeper to know on which region a row resides. The response of Zookeeper

is then cached to avoid the communication overhead. ZooKeeper holds a file called *hbase:meta*. This file contains informations about the regions such as the RegionServer location and the key range handled by this one.

3.4.3 Cluster replications

In HBase the data can be replicated on multiple HBase clusters [19]. In addition to propose a solution for crash disaster recovery, it's a way to copy the data between clusters that have different roles (e.g. the first serves the user requests while the other is used to apply map-reduce over data). The clusters could be located in different datacenters. The replication strategy consists in a master-slave architecture. The data on the slave servers is not synchronized with the master at each instant. Bellow is described the asynchronous protocol which leads to eventual consistency of the slave cluster.

Basically, the RegionServer of the master cluster keeps track of the modifications brought to the tables thanks to the WAL. Because the WAL contains the modifications (WALedits) brought to the tables, it is suitable to replicate the modifications on the slave cluster. The protocol is made of three steps:

1. The WALedits are filtered (e.g. filter by table) before being added to a file.
2. When the file reaches a specific size, it is sent to the slaves.
3. The file will be delivered to the slaves to be replayed.

The WAL is kept by a regionServer as long as it could be needed by the replicas. It is kept in the HDFS. The term "sent", doesn't mean that the files is sent. Actually, just the informations about the log files is sent to the slaves. The slaves can access to the log file via request on the HDFS. The master communicate these informations about files to the slaves via ZooKeeper.

ZooKeeper use three types of znodes to represent these informations:

- **state**: This znode indicates if the replication system is activated. This value is checked by the replication operations before going further.
- **peers**: This znode has one child znode for each connected slave.
- **rs**: This znode has three levels of children znodes (see figure 3.7).
    1. One child znode for each regionserver (A on the figure).

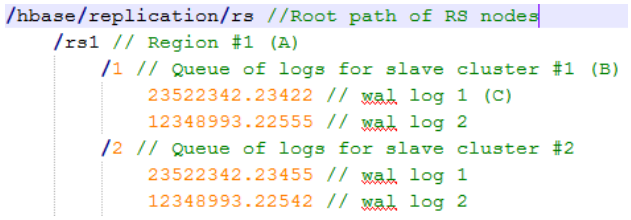

Figure 3.7: ZooKeeper organisation for cluster replication

2. Under these previous znodes, there is a znode for each slave cluster (B on the figure).
3. The last level (C on the figure) contains one znode by Log file to play. Each of these znodes contains a value which indicate the next WALedits to replicate. These znodes are sorted by creation date in order to apply modifications in the same order than on the master.

HBase proposes three modes of cluster replications:

- Master-Slave: The master pushes the modification to the slave cluster(s). It's important to understand that a slave cluster can own other tables for which it's the master.
- Master-Master: The two cluster are considered as master and can handle modifications on the tables (same or different tables). The modification are pushed inn both directions.
- Cyclic: In this configuration, we have a combination of the two previous configurations on more than two clusters. The master-master is a special case of cyclic configuration where the two clusters are already in master-slave configuration. Its important to be careful with loops when designing such configuration. Actually, loop detection is not guaranteed.

3.4.4 Timeline-consistent High Available Reads

The previous section presented a way to replicate data across multiple clusters. In this section we will present the read high availability using timeline consistent region replicas feature [20, 19]. Read high availability allows HBase

clients to perform reads when a regionServer has crashed. Therefore, that impacts the consistency of the read operation (Cap theorem). Actually, with this feature, the data that your read are not guaranteed to be up-to-date in some cases (see bellow).

The mechanism consists to create replicas of a single region on multiple RegionServers. The replication of the regions is under the responsibility of the MasterServer. The MasterServer will assign the RegionServers on which the replicas will be done (by default the number of replicas is set to 1). The LoadBalancer ensures that the region replicas are not located on the same RegionServer and rack (if possible). Each replica of a region has a replica id (which always begins at 0). We distinguish two kinds of replicas. The replica with the id 0 is called the primary region, the others are called secondary regions. Only the primary region can accept a write request, the others are read-only, making the write operation not highly available. The writes are send asynchronously to the others replicas by using the mechanism described in the previous section (3.4.3) about cluster replication. An interesting point is that the replicas don't use extra storage because they are located on the same cluster and therefore share the same HDFS. Only the memory stores could contain duplicates of the same data.

When replicas are activated, HBase can be configured on two level of consistency.

- The first is strong consistency, in this case the system only reads on the primary region (as if there were no region replicas).

- The second is the timeline consistency, in this case the system reads on the different replicas. Because data is not synchronized the consistency of the system is affected. With timeline consistency (see figure 3.8), a read RPC (remote procedure call) is sent to the primary server first. After a short interval, additional read RPCs will be sent to the replicas (if the primary has not yet responded). The read result will be the first response received. If the response came from a secondary server, the response might not be up-to-date (Result.isStale() return false if the response is maybe not up-to-date).

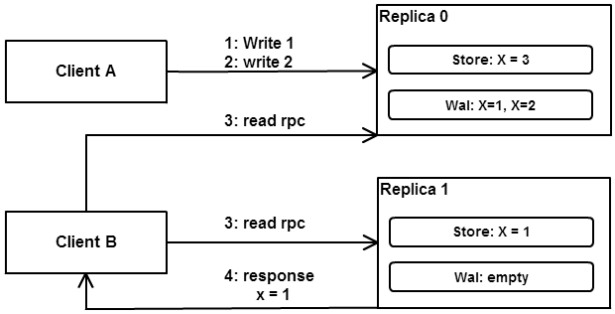

Figure 3.8: A timeline read example

3.5 Comparison Hbase vs. Cassandra

Cassandra and HBase present two very different solutions to achieve a distributed database able to store tera bytes of data. Selecting Cassandra or HBase depends of the requirements of the application which runs on top. This section proposes some key points depending on the use case.

The most important difference between HBase and Cassandra concern the approach of the CAP theorem. Cassandra has chose availability rather than consistency (Available-Partition tolerant system) while HBase has chosen consistency rather than availability (Consistent-Partition tolerant system). HBase provides transactions and guarantees ACID properties at row level (when ignoring high-available read). At a row level means that only modifications on a single row guarantees the ACID properties. Verifying consistency rules over the whole database is possible but would lead to poor system's performance. The drawback of this approach concerns the availability of the system. When a RegionServer fails, all the requests affecting its data rows will fail (read, write, etc.). Therefore, the system is not highly available. Although, we described some solutions to improve system availability (read high-availability and masterServers high availability), the system can't be highly available due to its initial design choices.

At the opposite, Cassandra has chosen availability in the CAP theorem. Although, consistency level of the request could be configured (one level, quorum level, etc.), Cassandra can't reach the same consistency level than HBase. This is mainly caused by the fact that Cassandra has no transaction mechanisms. The advantage of this approach is than the system can be highly available if replications and request consistency have been configured

in this direction.

Chapter 4

Stream processing

4.1 Storm

4.1.1 Introduction

Storm[21] is a real time fault-tolerant distributed stream data processing system which is one of the most used today. Storm was acquired by twitter in 2011 and became a great source of research in the stream processing domain. The fundamental objective of this kind of system is to make complex calculations in real time. This kind of computations is very common in applications such as twitter where each tweet is a source of many decisions for the system.

Common use cases for Storm are bigdata data-analytics, log-monitoring, behavior tracking etc. Actually it allows to perform the same computations a Map/reduce framework. Although, it is focus on the real-time aspect in order to achieve low-latency for real time events. For instance, Storm allows to compute tweets at twitter in order to extract the hashtags, references, etc. in order to reference these ones on the web platform.

Storm was designed to fulfill the following requirements:

- **Scalable**: the system must be able to add or remove nodes without disrupting the service.
- **Resilient**: the cluster must be able to continue the stream processing even if node crashes happen.
- **Extensible**: Storm must be able to communicate with other services to make the framework generic (e.g. communication with mysql to

compute on the twitter social graph).

- **Efficient**: when discussing about real-time services, we cannot ignore the system's performance. Storm must include various techniques to make the services as efficient as possible.
- **Easy to administer**: since Storm issues directly affect user experience, Storm must provide tools to point out the source of problems in order to solve the issues quickly.

4.1.2 Data model

The common way to understand how Storm process the messages is to think about the cluster as a topology which is designed by the developer in order to solve a particular problem. Topology can be seen as a directed graph on which *tuples* flow along the edges and are computed at the vertices. There are actually two kind of vertices, the *spouts* and the *bolts*. A spout produces a *stream* (an unbounded list) of data tuples. This data can be pulled from a queue such as Kafka or Kestrel. A bolt processes the incoming tuples and produces new ones (unless it's an end vertice) which are emitted to the next bolts in the topology. A topology can have cycles.

4.1.3 Shuffling

Storm provides different partitioning strategies to transfer tuples from a node to another.

- **Shuffle**: randomly distribute the stream to the consumer nodes.
- **Fields**: the stream is distributed to the consumer nodes according to a subset of the tuple attributes.
- **All**: the stream is replicated to all the consumer nodes.
- **Global**: the stream is sent to a single consumer node (the one with the lowest id).
- **Local**: The tuples are send to downstream nodes located in the same executor [22] (i.e. on the same machine).

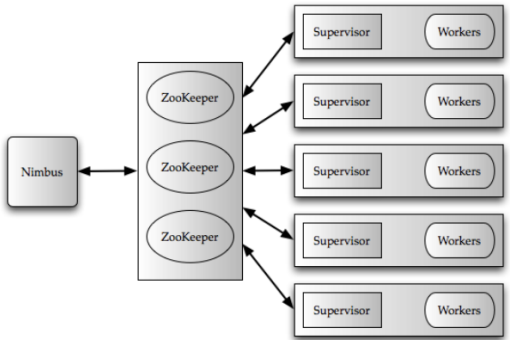

Figure 4.1: Storm overview architecture, taken from [21]

4.1.4 Storm key components

Storm runs thanks to three types of layers (each composed of one or more nodes) (show on figure4.1). The first is the *Nimbus* which is responsible to monitor the worker nodes and to create topologies. The second is the *ZooKeeper* layer which is responsible of storing the cluster state and to share information between Nimbus and workers. The third is the *worker* layer on which the topologies run.

Nimbus

The *Nimbus* node has the responsibility to assign the topologies to the worker nodes. The Nimbus is an Apache Thrift service [23] which allows the user to define topology in many languages thanks to Thrift objects. Apache Thrift allows "to define data type ans service interfaces in a simple definition file" (definition from apache doc [23]). In addition to the Thrift objects (which define the shape of the topology), the user submit the workers' code to the Nimbus via JAR files. The topology information is stored on disk (jar files) and on ZooKeeper (Thrift objects).

Once this data sent to the Nimbus, the Nimbus find available worker nodes to deploy the topology. To find these nodes the Nimbus observes the cluster state thanks to heartbeats. The coordinators (see below) send periodic heartbeats to the Nimbus to give two pieces of information. The first is that the worker is running; this information is used by the Nimbus to detect dead worker nodes. The second is about the server load to indicate if the worker node is able to run more topologies, this information is used by

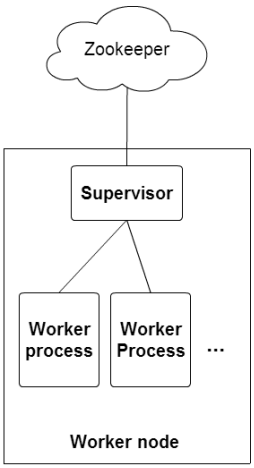

Figure 4.2: Worker node

the Nimbus to assign the pending topologies to the available worker nodes.

Zookeeper

Zookeeper is used to store information about the state of the cluster (such as the Thrift objects of the Nimbus) in addition to handle the coordination between the supervisors and the Nimbus. Storm relies widely on ZooKeeper, by using numerous mechanisms such as service discovery, ephemeral znodes etc.

Worker nodes

A worker node could be divided in two parts, the supervisor and the worker processes (see figure 4.2).

Each worker node has a Supervisor. The Supervisor receives assignments from the Nimbus and spawns the necessary worker processes and monitors them. The Supervisor is decomposed in four threads:

- **The main thread** bootstraps the node by reading the configuration, starts other threads, schedules timer events and stores local state in a persistent storage (such as hdfs).

- **The heartbeat thread** reports to the Nimbus every 15 seconds that the worker node is running.
- **The synchronize Supervisor event** is run every 10 seconds to handle the modification of the node's assignments. For instance, to detect that new tasks must be started, etc.
- **The synchronize process event** is responsible for the worker processes which run a fragment of the same topology (on the same node). It receives heartbeats from the workers and classifies them in four states: not started, valid, timed out or disallowed. For instance, the timed out state indicates that the process must be restarted, valid means that it runs properly, timed out that id didn't sent heartbeat and disallowed that it must be removed.

Many *worker processes* are running on the *worker node* at the same time. Each process runs several *executors* inside a JVM. Each executor runs several *tasks*, a task is an instance of bolt or spout. Each worker has four types of threads, the *worker receive thread*, the *worker send thread*, the *user logic thread* and the *executor send thread*. Send and receive threads are responsible for routing the incoming and outgoing tuples. In order to understand how the threads communicate, the following text describes the flow of tuples inside the worker node.

The receive thread listens on a TCP/IP port. When a tuple is received, the thread acts as a demultiplexor and pushes the tuple in the *in queue* of the right executor. The user logic thread of that executor gets the tuple from the in queue and runs the appropriate task to produce the output tuple. The output tuple is then pushed in the *out queue* of the executor. The executor's send thread pulls the tuples from the out queue and pushes them to the *global transfer queue* which contains the output tuples from several executors. Finally the worker's send thread analyses the tuples from the global transfer queue and sends them to the appropriate worker node.

4.1.5 Processing semantic

Storm provides two different levels of guarantees with respect to the tuple processing [24], *"at least once"* and *"at most once"*. The first relies on a system which ensures that each tuple in the stream had been processed at

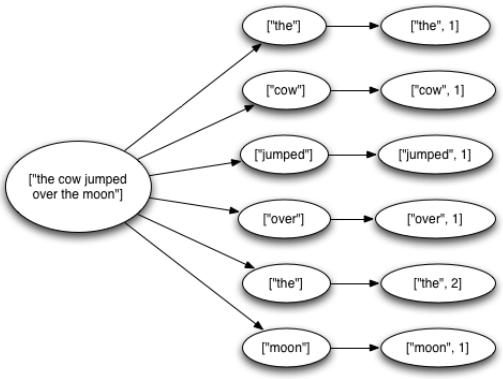

Figure 4.3: A tuple tree related to a word count, taken from [24]

least one time while the second doesn't give any guarantees about the tuple processing.

Figure 4.3 shows the tuples generated by a simple word count. A tuple coming out of a spout is considered fully processed if all the tuples of the tree (see figure) have been processed within a specified timeout. A tuple is considered failed when one or more tuples in the tree have not been processed within a specified timeout. By default the timeout is set to 30 seconds.

At least once semantic

To achieve the **at least once** semantic, Storm introduces a new kind of task in the topology, the *acker task* [24, 21]. This bolt tracks the state of the tuple tree which results from a *spout tuple* (i.e. a tuple received by a spout, on the left on the figure 4.3) from the processing of a spout tuple. The basic idea is to keep track of each tuple thanks to a 64 bit id. When the tuple emitted by the spout has been fully processed (resp. failed), the acker task sends an ack (resp. a fault) with the spout tuple id to the spout who emitted the tuple.

The spout tuple stays in the queue from which it has been pulled. In this way the message is removed from the queue only when acked or can be replayed by pulling the tuple again.

Now that we described "at least once" semantic guarantee inside Storm, the next problem to address is how to represent the tree in memory. A naive implementation could be to represent the tree with an n-tree but with complex topologies, a n-tree could become huge. Representing such tree a

Figure 4.4: Tree lineage, taken and modified from [24]

for thousands of spout tuples would lead to memory overflow.

To address this problem, Sotrm's solution adopted is to store a map at the acker-bolt which contains two values for each key (key => (value1, value2)). The key is the id of the spout tuple. The first value is the id of the spout which emitted the tuple and the second is a 64bit value called "ack value". When the spout tuple is emitted, "ack value" contains the 64bit id of this spout tuple. After it has been computed by the first bolt, one or more output tuples are created. Each of these tuples will have their own 64bit id. When the output tuples have been pushed to the queue, the bolt sends a message to the acker bolt which contains the id of the spout tuple (to get the key of the map), the ids of the output tuples and the id of the input tuple (the tuple which has been successfully computed). The input tuple id and the output tuples ids are then xored with the "ack value" (the value contained in value2 in the map, see above). In this way we follow the schema of the figure 4.4. Initially "ack val" contains C id. After the first bolt C, D, E ids are xored with C id. The xor results is the xor of D and E (because xoring the same id two times gives 0 in an idempotent fashion). In this way, we keep track of the tree with only 20 bytes in the acker task. The spout tuple is acked if the "ack value" reaches 0 (i.e. the tree has been fully processed). To resume the sequence is the following:

1. ackValue = C
2. ackValue = C xor C xor D xor E = D xor E
3. ackValue = D xor E xor E = D
4. ackValue = D xor D = 0

Failure scenarios

In case of failure of some part of the system, the "at least one" semantic must be always guaranteed. The following scenarios show how Storm continues to provide the guarantee that trees will be fully processed:

- **Spout task dies**: The queue from which the spout pulls the tuples detects the client loss and all the pending messages are pushed again on the queue. In this way, the tuples can be pulled by the available spouts.
- **Worker task dies**: In this case for each tuple lost because of the crash the timer associated with the tuples (located at the root of the topology) will time out. For each tuple, when its associated timeout elapses, it is replayed.
- **Acker task dies**: Each tuple tracked by the task will time out at the spout and will be replayed. As for any other task, the acker task will be recreated on another node.

At most once

This semantic gives no guarantee about tree processing. In this configuration we have no acker task. In this case, if a tuple is lost it is never be recomputed.

4.1.6 Faults tolerance and Storm resiliency

In the subsection 4.1.4 we described the components of Storm and their roles in Storm's architecture. Now an interesting point to consider is the behaviour of the system when faults occur. This section present some mechanisms used by Storm to handle faults in order to make the system resilient.

Nimbus

The Nimbus is a kind of master in the system which is not replicated. When the Nimbus crashes, it cannot continue to monitor the worker nodes, receive new topologies, reassign the work of the failed worker nodes and so on. We mentioned previously that all the information which represents the state of the Nimbus is stored on persistent storages (local file system or ZooKeeper). In this way the Nimbus can be revived and be immediately operational with

respect to the current topologies which run on the workers node. The Nimbus is fail-fast [24], meaning when an unexpected situation occurs the process is restarted. Stateless and fail-fast are the keys of the Nimbus resiliency. The Nimbus cannot be considered as a single point of failure because when it crashes the system is still running and only part of the service is unavailable, so the crash doesn't lead to the unavailability of the entire system. Actually, some project try to develop the high availability features for the Nimbus in best effort but none of these ones is in production. Actually that means that high availability for Nimbus is not very critical for the Storm's use cases.

Worker nodes

The worker nodes have to handle faults of the Supervisor and worker processes.

The first fault to consider is the crash of the Supervisor. When the Supervisor has failed, it can't continue to send heartbeat to the Nimbus. If a time-out occurs at the Nimbus (because the Supervisor has not sent heartbeat for a too long time), the entire node is considered as crashed by the Nimbus and all the tasks assigned to it are reassigned on the others worker nodes.

The second fault to consider is the crash of a worker process. As said in section 4.1.4, worker processes are monitored by the Supervisor which is waiting for their heartbeats. The Supervisor classifies the workers process in four categories: not started, valid, timed out or disallowed. The timed out category indicates that the node doesn't respond and is considered as dead. In this state, the Supervisor stops and respawns the worker process.

4.2 Conclusion

We presented Storm, a framework to build topologies which compute streams of data in real time. Its capacity to deal with large streams of data in real time makes a distinction with the batch processing framework (e.g. Spark presented in the appendix A). To be able to cover this use case, Storm had to provide a relaxed computation semantic, the at least once semantic. Although, at most once is also supported and most efficient for some use cases, at least once is a minimum requirement for many systems. We also discovered an excellent way to make the tasks (spouts and bolts) fault tolerant by

using a Supervisor which monitors the processes. In this way, if the Nimbus is running (Nimbus is the only single point of failure), Storm can guaranty that all the tasks are also running.

Chapter 5

Publish Subscribe

5.1 Introduction

In this chapter, we will provide an introduction to publish/subscribe through two practical technologies, Kafka an RabbitMQ. They are two important actors in the domain of the messaging systems and are widely used in production.

Since RoQ is a messaging system, this chapter will discuss about the problems met in such systems and their solutions from Kafka and RabbitMQ.

5.2 Kafka

5.2.1 Introduction

Kafka [25] development had begun at Linkedin in July 2011 and has been released under Apache License V2 in 2012. Kafka is not a traditional publish/subscribe service. It has been widely influenced by log aggregator and messaging systems in order to produce a very specific design which takes in the strengths of both. A log aggregator is a service (for instance Flume) used to collect efficiently a large amount of log data. This aspect of Kafka had been decided in order to provide a log aggregator able to deal with real time application [25]. To achieve these use cases, Kafka satisfies the following requirements [26]:

- High message throughput.

- Message persistence.
- Large backlog.
- Horizontal scalability.
- Support of multiple consumers.
- Automatically balanced system.

5.2.2 Overall architecture

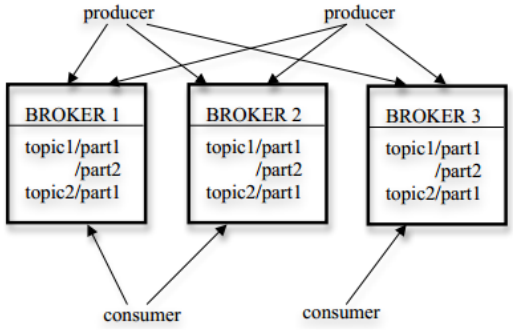

Figure 5.1: Overview of Kafka architecture, taken from [25]

Topic based messaging

At the user side, Kafka is made of producers and consumers. At the cluster side, Kafka is made of Brokers (see figure 5.1). Kafka is topic based. Each message send is associated to a topic. The consumers are pulling the data according to the topics they have subscribed. Meaning that the consumers is responsible for pulling messages, the Brokers don't push them to it as in traditional pub/sub systems.

The consumers groups

The consumers are regrouped in consumer groups. Such groups are made of consumers which are subscribed to the same topics. Each message are delivered at only one consumer of the group. In this way Kafka is able to provide

two services. A queue service, in this case the messages are spread among the consumers which belong to a user-group, in addition to a broadcasting service in this case all the consumers belong to a different group in order to receive all the messages.

Log partitioning

At the cluster side, messages are stored in different logs. Each log contains the messages for a particular topic. A log is split in partitions which are composed of segment files. Segment files are stored on the disks and have approximately the same size. When receiving a message, Kafka appending this one at the end of the current segment file. Kafka use the offset of the message in the file as its id.

One key in the design of Kafka is that each consumer from a consumer-group consumes from different partition. Therefore there are as much as partitions than consumers into the largest consumer-group for a given topic. In this way, Kafka is able to provide ordering guarantees for each consumer about what it consumes, and a high level of scalability because the partitions are located on different Brokers.

Producers

Producers send directly their messages to the partition of their choice. To help the producer to find the Brokers which handle the partitions for a given topic, the producers can ask metadata to any Kafka nodes. This metadata contains information about the partitions's location. To improve the throughput of the messaging system, the Kafka client doesn't send messages one by one but buffers them to send a batch which contains many messages. The drawback of this approach is that if the producer fails the buffered messages that have not been sent are lost.

To fairly divide the metadata requests among the Brokers, Linkedin uses load-balancer between the producers and the Brokers.

Coordination

ZooKeeper is used to store the metadata which is used by the coordination algorithms of the Brokers. In this way, Kafka doesn't require a master which handles the Brokers. The metadata contains information about the cluster state:

- **The Broker registry**: Each Brokers owns a Broker registry in Zookeeper. It contains the set of partitions and the set of topics handled by the Broker. This znode is ephemeral, when the Broker dies (or is shutdown), the znodes are removed.
- **The consumer registry**: There is a consumer registry for each consumer group. It contains the topics followed by the the consumers groups and an entry for each partition which stores its current consumers. When a consumer disconnect from the service (due to shutdown, crash, disconnection, and so on), it lost its entries (ephemeral znode).
- **The ownership registry**: One for each partition, this registry contains the consumer id that currently consumes the partition. This znode is ephemeral, when the Broker that owns the partition is disconnected, the znode is removed.
- **The offset registry**: Each partition has an offset registry. It contains the id of the last consumed message. This znode is persistent.

Algorithm 1: rebalance process for consumer C_i in group G

```
For each topic T that C_i subscribes to {
  remove partitions owned by C_i from the ownership registry
  read the broker and the consumer registries from Zookeeper
  compute P_T = partitions available in all brokers under topic T
  compute C_T =  all consumers in G that subscribe to topic T
  sort P_T and C_T
  let j be the index position of C_i in C_T and let N = |P_T|/|C_T|
  assign partitions from j*N to (j+1)*N - 1 in P_T to consumer C_i
  for each assigned partition p {
    set the owner of p to C_i in the ownership registry
    let O_p = the offset of partition p stored in the offset registry
    invoke a thread to pull data in partition p from offset O_p
  }
}
```

Figure 5.2: Rebalance algorithm, taken from [25]

The consumers register a watcher (see section 2.2.3) on the Broker registries (the registry of the brokt from which it consumes) and the consumer

registries at which it belongs. When a modification occurs, the consumer receives a notification and executes the rebalance process described on the figure 5.2. At the end of the algorithm, each consumer is assigned to a new partition for each topics that it follows. However sometimes, to ensure this result, the algorithm must be executed more than one times. This is due to the fact that consumers can concurrently executes this one. Therefore, the consumer can tries to take the ownership of a partition already owned by another consumer. In this case, the first one releases all its partition, wait a short period, and reexecutes the algorithm.

5.2.3 High availability

Replication strategy

Kafka introduced a replication strategy able to improve the availability of the system [26]. The mechanism replicates the partitions on different Brokers (i.e. nodes). Each replica maintains its own copy of the log.

We distinguish two types of replica, the leader and the followers. The leader is elected by ZooKeeper. Producers (resp. consumers) pushes (resp. pulls) messages to the leader. When a publisher pushes a message it receive the acknowledgement when the leader has propagated it to all the replicas in order to replicate it. With this approach the system can support n failures without loosing messages (actually n+1 is the number of replicas in the ISR set, see bellow). One drawback with this approach is that the leader can wait indefinitely when it propagates the message because one of the replicas doesn't acknowledge. Kafka tackles the problem thanks to the in-sync replica set.

The in-sync replica set (ISR)

Kafka builds a set called in-sync replica set (ISR). This set includes the replicas alive and up-to-date (i.e. which are in the same state than the leader). Initially, all the replicas are into the set, then the leader propagates the messages received to the replicas of the ISR set. If a replica fails, it is dropped out of the set. The leader maintains a value called high watermark (HW) which indicates the offset of the last committed message. This value is propagated to replicas. When a replica receives the HW, it flushes the data (to an offset greater or equal to HW) on the disk and copy the HW. The HW

is used to recover from crash (see section 5.2.3).

Consistency vs. Availability

The system offers to the clients a tradeoff between consistency and availability. The system can be set up to consistency, in this case after a message has been sent by the client, this one is acknowledged by the leader when all the followers have copied and acknowledged the message. Thereby no messages can be lost but the system latency increases. If the user chooses the availability, the message sent is acknowledged when it has been appended to the leader log, but before it has been acknowledged by the followers. In this case, the system latency decreases but messages can be lost.

Crash recovery

This section presents different failure scenarios and shows how Kafka deals with them [27].
When a *follower node fails*, it does three steps to recover:

- It truncates the message from the log with an offset higher than the water mark.
- It pulls the messages missed during the downtime.
- When the node is up-to-date, it joins again the ISR set.

When the *leader node fails*, we must consider three cases:

- Leader crash before writing the message to the log.
  Client will time out and will send again the message to the new leader.
- The leader crash after writing the message to the log but before responding to the producer.
  First, we must ensure that the message is propagated to the followers atomically (atomic broadcast).
  Second, the client after time out will resend the message. The new leader must ensure that the message will not be wrote twice.
- The leader crash after sending the response.
  Most simple case, a new leader is elected and the service continues.

To *elect a new leader* the mechanisms are the following:

- The alive replicas from the ISR register them-self to ZooKeeper.
- The first registered replica become the leader, it chooses the last message offset in its log as the new water mark.
- If the replica becomes a follower it truncates the registered entries with an offset greater than the hw.
- Each replica registers a listener to ZooKeeper to be aware when a new leader is elected.
- The leader waits for the new configuration to be completed (there is a maximum configured time). Then it registers the current ISR in ZooKeeper and begins to receive and send messages.

5.3 RabbitMQ

5.3.1 Introduction

RabbitMQ is a complete implementation of the advanced message queuing protocol (AMQP) and is programmed in Erlang. AMQP is an application layer protocol for messaging systems. RabbitMQ is distributed under Mozilla Public License by VMWare. It gives lot of freedom to the users which can make lot of choice at different levels [28]. It lets them trade off performance with reliability guarantees such as persistence, high availability, delivery semantic and so on. Users can define their own routing rules (from simple to very complex). It supports lot of messaging protocols although the core has been originally developed to support AMQP specifications. The client API is available for many language and operating systems. Many plugins are available and users can develop their own. To conclude, RabbitMQ allows users to make a messaging service which fit with their specific needs thanks to its high level of flexibility.

5.3.2 AMQP

Currently, RabbitMQ implements the specification V0.9.1 of the AMQP [29](V1.0 is supported by an experimental plugin) although it extends the protocol with

number of extensions. AMQP has been developed to define a protocol for messaging systems. It allows the clients to operate with any Broker which is conform to the protocol. Goal of such protocols is to remove vendor locking.

AMQP routing

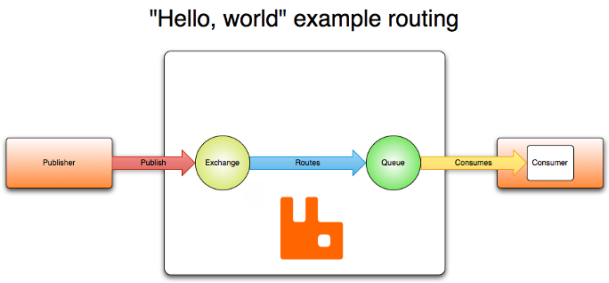

Figure 5.3: Simple routing schema with one Broker, taken from [30]

The Broker is made of three different entities: the Exchanges, the Bindings and the Queues. We can see on figure 5.3 how a message is routed from producer to consumer. First producer sends the messages to an Exchange which is responsible for placing the messages into the correct Queue(s), according to rules called Bindings. Finally, the Queue delivers the messages to the consumers which are subscribed to it.

AMQP is a programmable protocol, meaning that lot of decisions are taken thanks to the message's metadata.

Exchanges

AMQP defines different Exchange types which influences the routing algorithm (in addition to the Bindings):

- **Direct Exchange (default)**: The message is routed according to its routing key. The routing key defines the Queue's name in which the message should be placed.
- **Fanout Exchange**: This Exchange ignore the routing key and fanout the message to all the Queues bound to it.

- **Topic Exchange**: A pattern defines for each Queue what routing keys are bound to it. Thereby, the message can be routed to zero or many Queues.

In addition to the type, additional attributes define the Exchange: name, durability (allows the Exchange to survive to Broker restart), auto-delete (the Broker is removed when the Queues have finished to use it), etc.

Queues

The Queues store the messages destined to the consumers. Each message is delivered to one consumer. Thus, the messages are spread among the consumers (like in a traditional Queue). A Queue is defined by five attributes:

- **Name:** The name of the Queue in UTF8 characters (length up to 255 characters). If the application tries to create a Queue with a name which already exists, the system do nothing.
- **Durable:** Defines if the Queue must survive to a Broker restart. Doesn't mean that messages contained in the Queue are persistent. A non-durable Queue is called transient.
- **Exclusive:** The Queue is exclusively dedicated to a consumer. The Queue will be deleted when the connection has been closed by this one.
- **Auto-Delete:** The Queue is deleted when the last consumer has unsubscribed.
- **Arguments:** Additional arguments used by the Brokers (e.g. Queue TTL, etc.).

Bindings

The Bindings are the rules that are used by the Exchanges to redirect messages to Queues. For some Exchange types (e.g. direct Exchange), the Binding rules take an additional parameter called routing key (i.e. a topic). If a message cannot be routed to a Queue because no Binding matches with this key, the message is discarded or sent back to the producer (according to the settings).

Consumers

Consumers can consume messages in two ways. Via the pull API, in this case the consumer must explicitly request them by itself (as in Kafka). Via the push API, in this case the consumers are subscribed to the topics for which they are interested. Once subscribed, the messages are pushed by the Broker to them. Once that the consumer has finished its job, it can unsubscribe from the Queue.

Acknowledgements of the consumers can be used in order to handle consumers' crash. In this way unacknowledged message is resent by the Broker to another consumer, in order to guarantee that the messag is delivered at least one time to the consumer (actually, this feature is not sufficient to ensure at least one semantic, see section 5.3.4). In this case message a is removed from the Queue only when it has been acknowledged. It's also possible to remove the message after it has been sent (without ack). In this case the Broker cannot provide delivery guarantee.

Messages

Each message contains an header which defines some attributes, for instance: Content type, Content encoding, Routing key, Delivery mode (persistent or not), Message priority, Expiration period, producer application id, etc. Part of these attributes are used by the Brokers.

In addition to these attributes the message has a payload which contains bytes of data destined to the consumers.

As said previously the message persistence is not defined by the Queue attribute but by the message itself. The message is guarantee to survive to Broker crash only if the persistent attribute is set.

5.3.3 Distribute the Brokers

RabbitMQ proposes different ways to distribute the systems. We saw previously what are the components which compose the Brokers, now we see how we can distribute the Brokers.

Clustering

Clustering distributes the system by forming a Logical Broker with different Brokers. The Brokers are connected together through reliable links. In this

case, all the Exchanges are mirrored on each Broker, only the Queues are not mirrored by default.

Federation and Shovel

Shovel and federation are two different ways to achieve the same goal. This goal consists to allow the Brokers to define cluster topology. The Brokers can be located on the same or on different machines. Federation (and shovel) allows to assign routing policies among the Brokers. At a more fine grained level, messages are transmitted from upstream Exchanges to federated Exchanges. This solution allows to balance the workload of one Broker among several Brokers.

On figure 5.4, you can see that the producer can publish to the upstream Exchange, in this case, the message will be sent to the federated Exchange according to the defined topology.

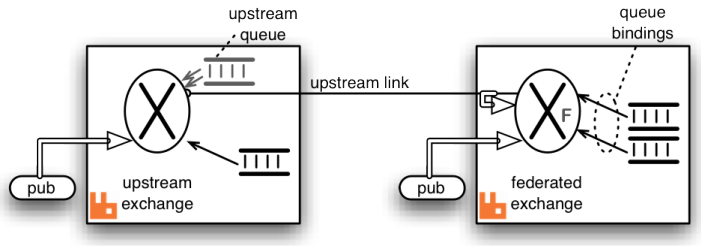

Figure 5.4: Federated Exchanges, taken from [28]

User can defines different topologies:

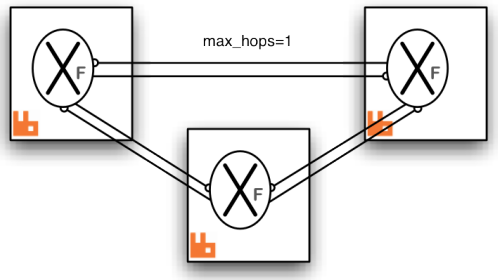

Figure 5.5: Fully connected topology, taken from [28]

- **Fully connected topology** (see figure 5.5): In this case the Broker which receives a message fanout it to all the others Brokers. The max-hop of the message must be set to one in order to avoid that the message loops in the Broker topology. In this configuration, the user can send its message to any Broker.

- **Tree topology** (see figure 5.6): The message is propagated from one root Broker to its children. The children send the received message to their children in order to propagate the message in the topology. The user has not to define a max-hop for this solution.

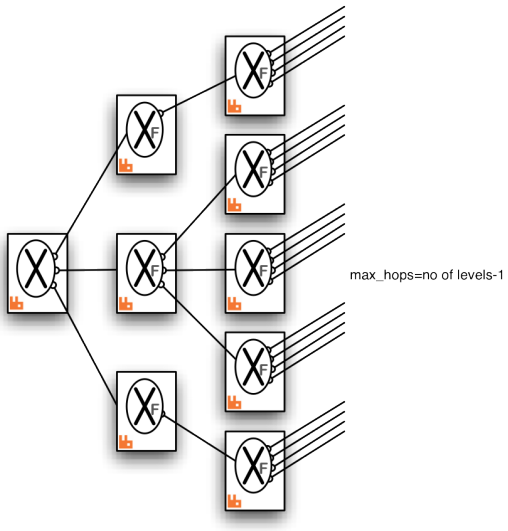

Figure 5.6: Tree topology, taken from [28]

- **Ring topology** (see figure 5.7): The Broker are connected together and form a ring (looks like a token ring topology). The message is send by the upstream Broker to its federated Broker. The max-hop must be set to the number of nodes.

Note that some topologies are not resilient, meaning that if one Broker fails, the routing could be compromised.

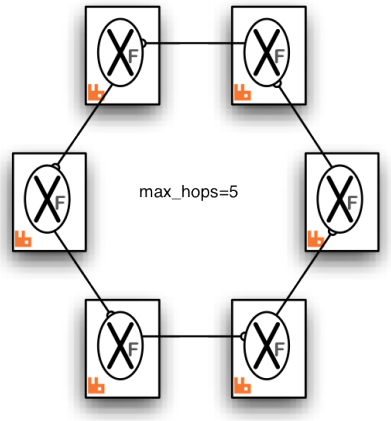

Figure 5.7: Ting topology, taken from [28]

Remarks

While this section aimed to give an overview of the AMQP protocol described in [30] [28], it is not complete. AMQP provides lot of freedom to users, therefore, lot of different mechanisms are implemented to improve the flexibility of the protocol. It's the reason for which it was not possible to cover all these ones in this introduction. The section provides a global overview of the implementation of AMQP's components in RabbitMQ and how they interact together in order to provide a messaging service.

5.3.4 High availability

This section present how RabbitMQ achieves high availability for its components.

Queues in RabbitMQ can be mirrored on the nodes which constitute the cluster [31]. A mirrored Queue consists of one master and one or many slaves. The master is elected thanks to a leader election mechanism.

Mirroring

Once again RabbitMQ propose different *policies* to elect the set of nodes on which the mirrors are installed.

- **All:** The Queue is mirrored on all the Brokers.

- **Exactly *n*:** The Queue is mirrored on exactly n Broker(s) (n is actually the Replication Factor). If n is set to three, the Queue will be mirrored on three Brokers. If there are less Brokers than n nodes, the Queue is mirrored on all the available Brokers. Once the nodes set, if one of them dies ant that the cluster has enough nodes, the Queue is replicated on a new node to maintain the Replication Factor.

- **Nodes (parameter: *node names*):** The Queue is mirrored on the nodes present in the *node names*) list. If all the nodes of the list are offline, the Queue is added on the node which has received the Queue creation request.

Queues synchronization

Slave can be *synchronized* or *unsynchronized*. In the former case, the slaves owns exactly the same set of messages as the master. In the other case, the slave's Queue has not the same messages as the master's one.

A slave can be unsynchronized for two reasons:

- When a new slave joins the mirrors set, its Queue is empty. From this instant all the messages received by the master are sent to the slave (asynchronously). The old messages - the messages received by the master before the slave existence - are not sent to synchronize the slave. Therefore, the slave is not synchronized with the master until the old messages has been acknowledged.

- When a slave has missed some messages (because node restart or network partition), it is not longer synchronized. RabbitMQ doesn't propose recovery mechanisms to get the missed messages. The only way to resynchronize the slave is to forget its old messages and to adopt the same behaviour as a slave which joins the mirror set (see previous point).

To ensure that Queues stay synchronized when these situations don't occur, RabbitMQ must ensure than the messages propagated from the master to the slaves are received by **all** the slaves in the **good order**. That lead you to the next section about the guaranteed multicast module [32] (also called atomic broadcast).

Atomic broadcast

Guaranteed multicast allows to create named groups which contain processes. At each instant, a process can join or leave the group. when a message is broadcasted to the group, it is guaranteed to reach all the group's processes during the lifetime of the message. The lifetime of the message is the interval between the instant when the message has been broadcasted and the instant when the broadcaster knows that the message has been received by all the processes. In addition, the algorithm provide local-ordering (i.e. if a process send M then M', the group's processes will receive M then M').

Master Failure scenarios

When the master fails, three situations are possible.

The first situation happens when the master owns one or more synchronized slaves. In this case a synchronized slave can be elected as leader without message loss.

The second situation situation occurs when the last master fails and that no slaves are alive. Because the last node that failed was the last master, RabbitMQ waits for it restarts to elect it as master. In this way, we avoid messages loss.

The third situation occurs when no slaves are synchronized with the master and that it fails. In this case user must choose consistency or availability:

- **Consistency (default):** The system doesn't elect a not synchronized mirror as leader, in order to avoid message loss. Therefore, the Queue is just shut down. In this way, the producers don't receive the acknowledgements and resend the messages until a master becomes available. The recovery mechanism follows the same path as the second situation.
- **Availability:** The system elects an unsynchronized Queue as master. In this case, some messages can be lost.

5.3.5 Message acknowledgement at the producer side

In this section we describe how the messages are acknowledged according to different configurations. When the message can be routed by the Exchange it sends the acknowledgement only when the message has been pushed in the destination Queue. If the Queue is mirrored the message is acknowledged

only when all the mirrors have acknowledged. Finally, if the message is set to be durable, the message is acknowledged only when it has been stored on the disk. In the other case, i.e. the message cannot be routed by the Exchange because no Queue matches, the Exchange sends a confirmation token. If the message is set to "mandatory", the Exchange sends a special token that indicates that no Queue has matched before sending the confirmation token.

5.4 Conclusion

RabbitMQ and Kafka are two different solutions to implement a scalable messaging service which provides some guarantees such as message persistence, *at least one* semantic, etc. We will see in the the next part of this master thesis that RoQ has a very different architecture which is not based on Brokers, however the problems met are the same. In this way, the solutions studied could be interesting for RoQ's problems.

Part III

High availability for RoQ

Chapter 6

RoQ

6.1 Introduction

This chapter first describes EQS's architecture, an architecture on which RoQ is based. Second, we describe RoQ's own goals and its current architecture.

6.2 EQS

6.2.1 Overview

EQS [33] has two types of clients: producers and consumers. Producers have the ability to publish messages in a logical queue. The queue then delivers the messages to its subscribed consumers. EQS aims to satisfy the following requirements:

- **Elastic scalability**: The maximum throughput increases linearly with the number of instances.
- **High throughput**: Large amount of messages transmitted by unit of time.
- **Integration on a cloud**: Designed to run on a cloud infrastructure.

6.2.2 Components

EQS satisfies this requirements by implementing the following components:

- **Exchange**: The component which routes messages fro producers to consumers. An Exchange is stateless, messages are transmitted thanks to their route keys. This property ensures that the queues can be scaled up just by increasing the number of Exchanges.
- **Logical Queue**: Logical concept which designates a group of Exchanges.

  that handle a common the entity which transmits the messages from producers to consumers according to a particular subject. The queue is composed of one or more Exchanges.
- **Queue Management**: Contains logical queue metadata such as name, url, physical location, list of Exchanges and Service Level Agreement (Parameters which define the minimum throughput that the queue must achieve).
- **Monitoring**: Component used by the system to get information about the health of the service. The Monitor keeps track of the Key Performance Indicators (KPIs). In addition, when an Exchange fails, it is responsible for relocating producers to a new one.
- **Rules and Scaling Management**: Thanks to the KPIs provided by the Monitor, the Scaling Management component is able to take decisions about whether to scale in (reduce the number of Exchange instances) or scale out (increase the number of Exchange instances).

6.2.3 EQS in action

Connecting Producers/Consumers

When a consumer or a producer wants to establish a connection with a logical queue, it contacts first the *Queue Management* component to get the address of a free Exchange. If the queue does not exist yet, a new Exchange is created and its metadata is registered in the Queue Management component. The Exchange is then registered in the *Monitoring* component.

Scaling

Scaling out is the operation which consists in increasing the number of Exchanges for a logical queue. This operation is performed to decrease the work-

```
Queue q;
if q.load > kpi_queue
  Exchange new_exchange createNewExchange();
  List<Exchange> L = q.getOverloadedExchanges();
  while(new_exchange.isNotOverloaded)
    for Exchange e in L
        AND e.getNbOfConnectedProducers > 1
      Producer P = e.getMostProductiveProducerID();
      P.relocate(new_exchange);
  q.getListeners.updateExchanges(new_exchange);
```

Figure 6.1: EQS Scaling algorithm, taken from [33]

load of the current Exchanges in order to improve the queue's performance in term of throughput, latency, etc. The *scaling algorithm* (see figure 6.1) is executed when one of the following situations occurs:

- The latency of message delivery is higher than the specified threshold.
- The throughput is behind the specified minimum throughput.
- The average number of consumers/producers connected to the Exchanges is higher than the specified threshold.

These situations are detected by the Monitoring component. The algorithm consists in relocating the current producers from the overloaded Exchanges to a new one.

Th opposite operation is *scaling in*: reducing the number of Exchanges for a logical queue. This operation can be performed for the following reasons:

- Resources are not used efficiently. There are too many Exchanges compared for the current workload. In this case, Exchanges are shut down to avoid wasting resources.
- In a pay-per-use model, it can happen that there are not enough credits to maintain the current level of service. In order to conserve credits, the SLA can include a rule to reduce the queue to the minimal level of service when this situation occurs.

6.3 RoQ guarantees

6.3.1 current guarantees

- **Scalability**: RoQ is horizontally scalable, if the cluster's workload increase we can add nodes to imrprove the system's performance.
- **Metadata reliably stored**: The metadata handled by the master is reliably stored in ZooKeeper.
- **Elastic scale up**: The number of Exchanges increase dynamically when the queu's workload is increasing. The Elastic scale down is not already implemented.
- **Exchanges are fault tolerant**: When an Exchanges has crashed, all the publishers and subscribers connected to it are relocated on an available one.

6.3.2 Missing features

- **High Availability**: When one process crash (excepted an Exchange), RoQ is not able to detect and recover it. Therefore, part of the service is down until it has been recovered manually. The goal of this thesis is to make RoQ fault tolerant, in order to improve the service availability.
- **Message persistence**: If a Subscriber is temporarily disconnected, it has no way of retrieving missed messages. In addition, if an Exchange fails, lost messages cannot be recovered.
- **Message ordering**: Since messages are spread among Exchanges and no coordination mechanism among them exists, messages cannot be ordered. The only way to guarantee an order between the messages send by a single publisher, is to send all of them to the same Exchange. In should be noted that most messaging systems do not provide this guarantee (e.g. Kafka, NSQ).
- **Subscriber groups**:Messages are fanned out to all the Subscribers for a given topic.
- **Delivery semantic**: Currently RoQ provides no delivery semantic for the following reasons:

- Message duplication is possible. When a publisher send a message to an Exchange, the acknowledgement can be lost, in which case the publisher sends the message once more. Since the Exchange doesn't de-duplicate, messages can be delivered twice.
- Messages can be lost. Even though the Exchanges are stateless, they still buffer messages while transmitting them. Therefore, if an Exchange fails, all the buffered messages are lost.
- There is no consumer acknowledgement. Messages sent by Exchanges to consumers are not acknowledged. If a consumer crashes or temporarily disconnects it cannot retrieve missed messages.

We can conclude that no delivery semantic is provided. The "at least once" semantic cannot be guaranteed due to message loss and the absence of consumer acknowledgement. The "at most once" semantic cannot be guaranteed due to possible message duplication. By extension, "exactly once" semantic cannot be guaranteed.

6.4 RoQ: Architecture

6.4.1 Components overview

RoQ's architecture [34] (see figure 6.2) is made of four components. RoQ implements the same architecture as EQS, although some modifications have been brought.

The first is the *Global Configuration Manager* (GCM). It knows all the nodes of the cluster, i.e. the hosts and the ZooKeeper servers. It manages the cluster metadata and is contacted by clients to create or remove queues, to modify scaling policies, to establish connections with queues, etc.

Each host has a *Host Configuration Manager* (HCM) responsible for creating or shutting down processes (such as Monitors, ScalingProcesses etc.). HCMs manage the *Logical Queues* which are composed of *Monitors*, *ScalingProcesses* and *Exchanges*. The *Monitor* maintains the list of *Exchanges* for a particular queue, it contains a thread called *StatMonitor* which keeps track of resource usage (KPI). The *ScalingProcess* is responsible for executing the scaling algorithm according to the scaling policy. There is exactly one *Monitor* and one *ScalingProcess* per logical queue. The *Exchanges* are the

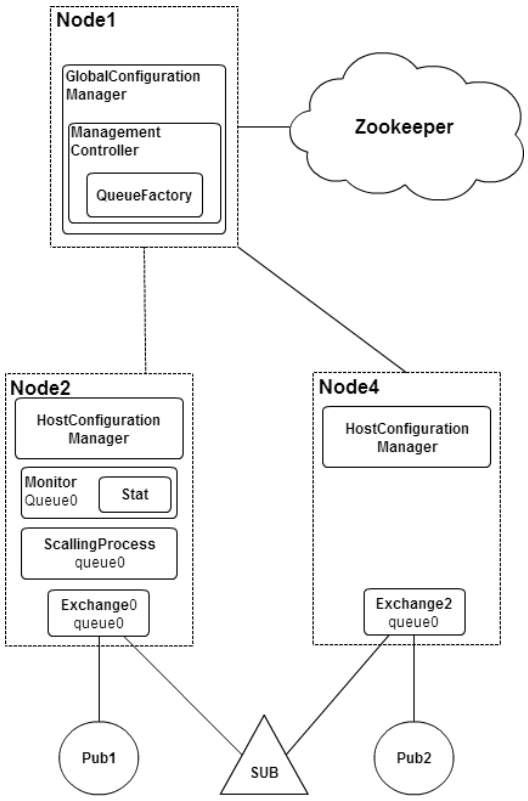

Figure 6.2: RoQ overview architecture, a cluster with one queue

connection end-points for the clients. They route the messages from publishers to Subscribers. There are one or more Exchanges per Logical Queue.

Interconnection between processes

RoQ processes communicate via ZeroMQ [35] sockets. ZeroMQ is a networking library which provides sockets able to send an receive messages over different transport protocols. This powerful and complete library allows abstracting the implementation from the network layer. It supports the following communication patterns:

- **Pair**: This is the simplest ZMQ socket. It allows two peers to communicate together. The messages are not sent and received reliably.
- **Request/Response**: The most classical pattern composed of two

types of ZMQ sockets. The first one is the REQ socket which is used by a client process to send a request to a server process who responds via its REP socket.

- **Publish/Subscribe**: With this pattern, Subscriber sockets are connected to one or more publisher sockets. The publisher fans out its messages to all the Subscribers.

- **Push/Pull**: With this paradigm a producer enqueues messages in its push socket which allows consumers to then pull the messages on demand.

One drawback of using ZMQ is that we cannot control all the technical aspects of the system's network layer. ZMQ's documentation is sometimes unclear about how the messages are handled by the sockets. For this reason, a study about how RoQ uses ZMQ sockets should be done in order to define RoQ's reliability.

6.4.2 Publishers

Each publisher sends its messages to its assigned Exchange. Each message sent to a queue is composed of a payload and a topic. The connection with the Exchange is established thanks to the following steps:

1. The publisher contacts the Global Configuration Manager to get the Monitor's address for that queue.

2. The publisher contacts that Monitor in order to get the address of an Exchange.

3. It establishes the connection with the Exchange.

Once connected, the publisher can send its messages to the Exchange. However, two events can occur:

- The publisher can disconnect from the Exchange.

- The publisher can receive a relocation notification in order to be assigned to a new Exchange.

6.4.3 Subscribers

In RoQ, Subscribers receive all the messages for the topic they have subscribed to. To register its interest for a given topic, the Subscriber establishes a connection with all the queue's Exchanges thanks to the following steps:

1. The Subscriber contacts the Global Configuration Manager to get the Monitor address responsible for that queue.
2. The Subscriber contacts the Monitor to get all the addresses of the Exchanges in that queue.
3. The Subscriber establishes a connection with each Exchange to receive the messages.

Once connected, it handles notifications from the Monitor which informs the Subscriber if a new Exchange is created on that queue. The Subscriber must then establish a connection with this Exchange in order to continue receiving all the messages. In addition, the Subscriber periodically sends throughput statistics to the StatMonitor (described in section 6.4.6).

6.4.4 Shutdown Monitors

A *Shutdown Monitor* is a simple thread bound to a ZMQ port. It is responsible for receiving shutdown requests from an other components in order to cleanly stop the process that spawned it. The following processes run a Shutdown Monitor:

- Global Configuration Manager
- Host Configuration Managers
- Monitors
- ScalingProcesses
- Exchanges

6.4.5 Exchanges

An *Exchange* is a connection end-point for publishers and Subscribers. It forwards the messages from its connected Publishers to its connected Subscribers. An Exchange is started by the *Host Configuration Manager*. In addition it is responsible for the following operations:

- Handles disconnection notifications from publishers.
- Periodically sends heartbeat to its Monitor via *HeartBeatTimer*.
- Periodically sends statistics about throughput and resource usage via *ExchangeStatTimer* to the StatMonitor instance of its Monitor.

The Exchange can be shut down thanks to its *ShutdownMonitor* module.

ExchangeStatTimer

The *ExchangeStatTimer* is a scheduled task which runs every minute, and which sends message throughput for an Exchange to the queue's StatMonitor.

HeartBeatTimer

The *HeartBeatTimer* is a scheduled task which runs every ten seconds, and which sends a heartbeat message to the queue's monitor. This heartbeat allows the Monitor to detect failed Exchanges.

6.4.6 Monitor

A monitor handles a queue and is started by its HCM. Publishers must contact it to be assigned to an Exchange on which they can send messages. In the same way, Subscribers get the addresses of the queue's Exchanges from the Monitor. The publishers and Subscribers maintain a constant connection with the Monitor in order to receive its notifications.

The Monitor is also responsible for monitoring its queue's Exchanges by performing the following operations:

1. Relocate Publishers which are connected to a dead or overloaded Exchange.
2. Notify the Subscribers when the list of Exchanges is modified.

StatMonitor

The Monitor runs a *StatMonitor* responsible for centralizing statistics about its queue. These statistics are based on the following sources:

- Exchanges metadata sent by the Monitor.
- Throughput statistics sent by Exchanges and Subscribers.

6.4.7 ScalingProcess

Each queue owns a ScalingProcess instance. At start up (see figure 6.3), the ScalingProcess gets the queue's *StatMonitor* address from the *Management Controller*. It uses this address to receive KPIs from the StatMonitor. It then gets the scaling configuration from the *Management Controller* (see section 6.4.8).

Once started, when it receives a KPI update, the ScalingProcess executes the scaling algorithm according to its scaling policy. This operation can lead to the creation or destruction of Exchanges.

Currently, RoQ implements only the scale out algorithm.

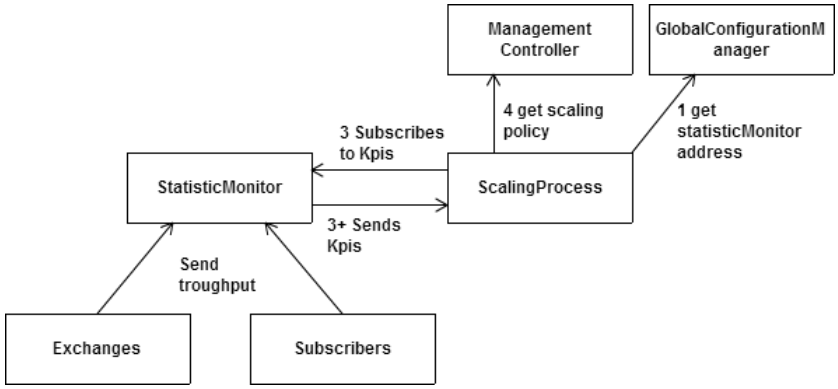

Figure 6.3: RoQ kpi subscription

6.4.8 Global Configuration Manager

The *Global Configuration Manager* (GCM) handles information provided by the HCMs such as host address, queue's metadata, etc. Hence, the GCM

knows the cluster topology. Clients connected to the GCM can perform the following:

- Get addresses of *Monitors* and *StatMonitors*.
- Get the cluster topology, such as the hosts list.

The GCM only handles metadata storage in ZooKeeper. It is not responsible of operations such as queue/host creation. The GCM periodically publishes the cluster topology via the *GlobalConfigTimer* instance.

Management Controller

At start up, the GCM spawns an instance of *Management Controller*. While the GCM performs operations on the metadata storage, the Management Controller performs the actual creation/destruction of queues and Exchanges via a *LogicalQFactory* instance.

GlobalConfigTimer

This scheduled timer periodically sends topology information to connected third-party clients.

6.4.9 ZooKeeper

Zookeeper is the RoQ metadata storage. Originally, RoQ used a SQLite database. However SQLite was removed in view of implementing high availability. Currently ZooKeeper stores the following metadata:

- **Queues**: Queues' metadata such as their names, Monitor addresses, StatMonitor addresses, host addresses and their states (running or not).
- **Scaling policies**: the description of the auto-scaling policy:
  - Exchange rule: maximum throughput.
  - Host rule: maximum memory usage, maximum CPU usage.
  - Logical Queue rule: maximum throughput per Exchange, maximum producers per Exchange.
- **Connected hosts**: Addresses of the connected hosts.

- **Cloud configuration**: Cloud credentials for infrastructure management.

6.4.10 Host Configuration Manager

There is exactly one Host Configuration Manager (HCM) per host. An host is a node which runs Monitors and Exchanges. At start up, the HCM sends a message to the GCM in order to be registered on the cluster. The HCM provides an interface for the other components, which allows them to do the following operations:

- Create a queue. This operation consists in creating all the components (i.e. processes) required to run a queue: a Monitor, an Exchange and a ScalingProcess.
- Remove a queue. This operation consists in shutting down all the processes used by a queue by using their Shutdown Monitors.
- Get the current and maximum number of Exchanges on this machine.
- Create an extra Exchange.

Chapter 7

Abstracting solutions

7.1 Introduction

In this chapter, we will formalise the problems described in the state of the art.

Since this master thesis concerns the high availability for a messaging system which has a master/slave architecture, we will study the solutions from the state of the art which are in relation.

First, we present the high availability solutions from Storm and HBase. Second, we study the delivery semantic solutions provided by RabbitMQ and Kafka, although we will not implement reuse them in this thesis.

7.2 High Availability for Master/Slave architectures

We designate by master slave architecture, an architecture where a particular component called master is responsible for its slaves. In this section, we will abstract the high availability problematic for master and slave separately. In each case, we will illustrate the problem ans its solution through practical examples from the state of the art, then we will generalise the solutions.

7.2.1 Master availability

HBase: HMaster

To implement this solution, HBase rely widely on ZooKeeper.

- HBase runs the same master process on different nodes.
- One of these processes is elected as active and the others as passives by the leader election mechanism of ZooKeeper. When the active master dies, it stops to send heartbeat to ZooKeeper which elects a new active master automatically.
- ZooKeeper is also used by the active master to store metadata. In this way, the active and the passives nodes are stateless and always synchronized because each master has the access to ZooKeeper. The system can rely to it because it is garant of the persistency of the master data.
- Only the active master can perform update operations on the ZooKeeper znodes.

Storm: Nimbus

The Storm Nimbus presents closely the same characteristics. Therefore, the way to make it highly availables is the same. First, it is important to notice that currently the high availability for this component is not implemented in the current Storm release. However, a solution has already been proposed. The following points describe this one.

- Storm runs the Nimbus process on different nodes.
- One of these process is elected as the active thanks to ZooKeeper (as for HBase).
- This is currently the most important problem. Although some critical data are stored on ZooKeeper, some others are present on the local disk of the node which runs the Nimbus process. The information on the disk must be the same for all the processes. The solution proposed for this open issue should be to store this information on a distributed file

system such as HDFS. In this way, all the master will share the same state. In my opinion, this is the most suitable solution to tackle this problem (https://issues.apache.org/jira/browse/STORM-166).

- Only the active Nimbus is able to modify topologies information in ZooKeeper and on the HDFS.

Abstraction

The **problem** of the availability for the master is the following:

1. The master is always running.

The **solution** for the master is the following:

1. One and only one master process is active.
2. When a master fails, we detect the failure and we activate another one.
3. Each master process is consistent.

7.2.2 Slave availability

Storm: Worker nodes

The worker nodes are the slaves in the Storm architecture, they must run the tasks that compose the topology. These ones are instances of spouts or bolts. For this solution, we can decompose the problem in two abstractions. The first is the worker nodes' tasks monitoring by the supervisor. The second is the supervisor monitoring by the master. Together, these two abstractions will satisfy the given solution.

Abstraction 1, tasks monitoring by the supervisor (ii):
The supervisor receives tasks from the master. It must run and monitor these tasks on its local executors. These tasks send periodic heartbeat to the Supervisor. If this last one doesn't receive the heartbeat in time, the task is timeout - meaning considered as death - and the supervisor reschedules it.

Abstraction 2, supervisors monitoring by the Nimbus:

- The Nimbus eventually plans each task on a worker node.
- It keeps the information about the tasks which run on each worker in a persistent way.

- The worker nodes register an ephemeral node in ZooKeeper ant the Nimbus watches it.
- When the node is dead, the Nimbus eventually detects the failure thanks to its watcher and re-plans the tasks from the lost node.

HBase: Region servers

The region servers are the slaves in HBase architecture. Each of them run several instances of region.

- Each region registers an ephemeral node in ZooKeeper ant the HMaster watches it.
- When the node is dead, the HMaster eventually detects the failure thanks to its watcher and re-deploy the regions from the lost node to new ones.
- The different regions runs on the same process which is the one which maintains the ZooKeeper session. If this one is alive, all the regions are alive.

Abstraction

The **problem** is the following:

1. All the processes are planned on the slaves.
2. These processes accomplish their function until they are stopped.

The **solution** is the following:

1. The master detects slave failure and restarts its processes.
2. The slave runs it assigned processes.

7.3 Delivery semantic

7.3.1 At most once

This is the most simple solution, because that doesn't provide a guarantee about message delivery. The **problem** is the following:

- A message from the producer is delivered at most one time to the consumer.

A **solution** to this problem is:

- The producer sends the message to the messaging system at most one time.
- The messaging system sends the message to the consumers at most one time.

It is not interesting to illustrate this solution with examples from the state of the art. The system has just to send its messages to consumers in best effort but these ones can be lost.

7.3.2 At least once

Unlike *at most once*, the *at least once* semantic guarantees that the messages will be delivered to the consumers at least one time (duplication is possible). We distinguish the Kafka's solution from RabbitMQ's, because we saw that Kafka uses the concept of consumer groups.

At least once with subscriber groups

- The master partition acknowledges the message after it had been replicated on the slaves.
- Kafka stores the messages on multiple nodes thanks to mirroring.
- The Rebalance algorithm ensures that the consumers of each subscriber group consume all the partitions and that a partition is consumed by one and only one consumer of each group.
- A consumer sends an acknowledgement when it received a message. Kafka stores in ZooKeeper the if of last message received by the consumer.

At least once without subscriber groups

RabbitMQ is a good candidate to provide a concrete solution for this model. Suppose that the brokers are distributed in clustering mode and the queues are mirrored (see section 5.3.4):

- A brokers send an acknowledgement only when all the mirrors have received the message.
- Each message is reliably stored on multiple mirrors.
- A message is removed from the mirrors only when it has been acknowledged by a consumer.

Abstraction

The **problem** is the following:

1. A message from the producer is delivered at least one time to the consumer.

A **solution** to this problem is:

1. The broker must sends and acknowledgement when a message has been received.
2. The messages must be persistent, meaning reliably stored on the cluster (mirror, HDFS, distributed database).
3. The consumer must indicate acknowledge the received messages.
4. (for subscriber group only) A message cannot be sent to two consumers of the same group (partitioning).

Chapter 8

High level solution for RoQ

8.1 Introduction

We saw in the previous chapter common problems related to high availability and their solutions. We will now propose a solution for the high availability of the RoQ core components, the main goal of this thesis. The proposed solution will combine the solutions from HBase and Storm since RoQ has a master/slave architecture. HBase proposed a solution to make the master highly available by using ZooKeeper and failover mechanisms. Storm proposed a solution to ensure that each node and its processes are monitored and run as expected.

its processes and to detect slave failure in order to restart its processes on another node. A the end of this chapter, the solution for high availability of RoQ will be defined and ready to be implemented.

8.2 Global Configuration Manager (GCM)

The GCM handles metadata storage in ZooKeeper. We saw two very similar components in the state of the art, the HBase master and the Storm Nimbus, although only HBase implemented a highly available solution for its master.

As a reminder, a highly available master implies:

1. The master is always running.

The **solution** for the master is the following:

1. One and only one master process is active.

2. When a master fails, we detect the failure and we activate another one.

3. Each master process is consistent.

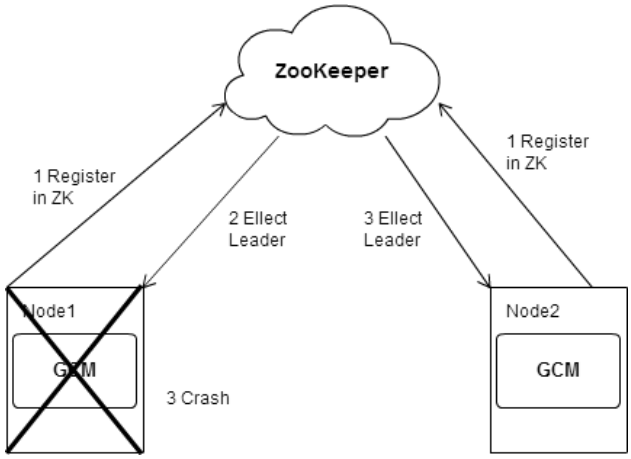

Figure 8.1: GCM failover mechanism

The HBase solution was based on the fact that master's data is stored in Zookeeper, in order to make the master itself stateless. This allows for shorter recovery time from failures.
ii + i) We can use the leader election mechanism from ZooKeeper in RoQ.
i) Metadata is currently stored in ZK and GCM responsibility is to handle this data. Therefore, if we deploy several masters, they all have access to the same data and all of them are in the same state.

8.3 Host Configuration Manager (HCM)

We have to manage the slave's problem. Actually, the HCM looks like the Supervisor of the Storm's slaves, it's a type of node manager.

Let's remember the definition for the slave:
The **problem** is the following:

1. All the processes are planned on the slaves.

2. These processes accomplish their function until they are stopped.

The **solution** is the following:

1. The master detects slave failure and restarts its processes.
2. The slave runs it assigned processes.

For RoQ, we must consider that **a task is a process**. Like for Storm's supervisors we decompose the solution in two sub-solutions:

1. The monitoring of Exchanges, Monitors and ScalingProcesses by the HCM.
2. The monitoring of the HCM by the GCM.

These sub-solutions will provide together the general solution described earlier.

In what follows, we will first describe the failure detection mechanisms, then the crash recovery mechanisms. In this way, we will cover the problem from failure detection to full recovery.

8.3.1 Failure detection

Exchange, Monitor and ScalingProcess monitoring by the HCM

The processes have to exchange heartbeats with their HCM to ensure that they run as expected. If one of them seems blocked, the HCM must be able to stop or kill it and replace it. This mechanism is described in figure 8.2 and was inspired by Storm.

HCM monitoring by the GCM

The GCM monitors the HCM via the service discovery feature provided by Curator, Netflix's ZooKeeper library. This feature monitors available services through the use of heartbeats. When an HCM joins or leaves the cluster, due to a crash or because it shuts down, the GCM receives a notification via this service. If the GCM detects that an HCM has failed, it must redeploy the lost Monitors and ScalingProcesses on the other nodes.

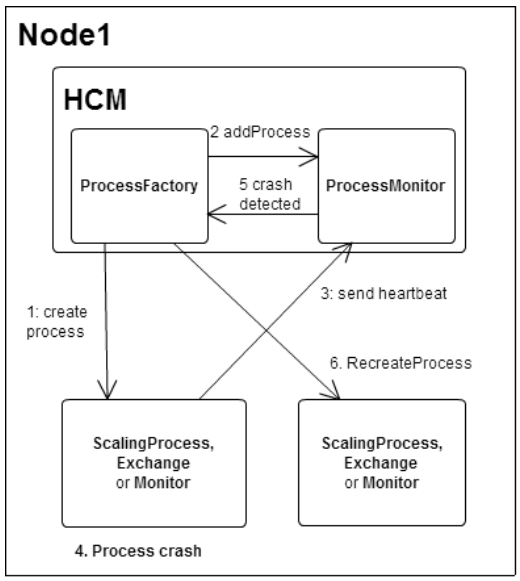

Figure 8.2: Exchange, Monitor or ScalingProcess crash recovery

HCM monitoring by a daemon

The two previous points have covered the general solution but a simple failure of the HCM process triggers a full node recovery. It is possible to avoid this overhead by deploying a small system component which monitors the HCM and which recovers it when it crashes. This way we could avoid the complete node recovery caused by an HCM crash. This point is optional but can decrease the down time, and thus improve system availability.

8.3.2 Process crash recovery

Once a process has been detected as crashed by its HCM, it is restarted. The HCM has the data to recreate the process properly. More details will be presented in the next chapter.

8.3.3 HCM crash recovery

This recovery process is the most sophisticated and is described in figure 8.3. In what follows, we first describe the Monitor replicas, then the node

recovery process. Finally, we treat two particular problems resulting from this solution.

Monitor replicas

High availability cannot be achieved with only one Monitor for each queue. The reason is that the clients (publishers and subscribers) receive information from the Monitor through a ZeroMQ broadcast socket. With this type of socket, the Monitor doesn't know which clients are subscribed, it simply sends its message to the socket and the information is multicast to them. In this way, if the Monitor is lost, we have no way of reestablishing the connections with the clients.

Since we cannot lose connection with clients, we must introduce *backup Monitors* which are also connected to them. Unlike the *active Monitor*, backup Monitors do not process requests, they only wait until they are activated by the HCM. In this way, if the active Monitor is lost, a backup Monitor is activated.

Recovery process

Thanks to backup Monitors, we can describe a **recovery process**. When the GCM detects that an HCM is dead, it fetches backup addresses for the lost active Monitors. The GCM then sends messages to the HCMs responsible for the backup Monitors in order to activate them. When an HCM receives this message it creates one ScalingProcess and one Exchange, then it activates the backup Monitor.

Replication factor

The *replication factor* corresponds to the number of replicas for a particular Monitor. For instance, a replication factor of three implies that we have one active Monitor and two backups.

We must ensure that the replication factor is always enforced. Hence, when an HCM has been detected as dead, the GCM also recreates a backup Monitor for each Monitor (active or backup) which was present on the dead node.

The reviving node problem

The last problem to solve is the **reviving node problem**: if an HCM is considered as dead by the GCM while it has not crashed, the recovery process will duplicate Monitors, ScalingProcesses, etc. This situation can happen when a network partition has occurred or when an HCM is overloaded. To avoid this problem, when a node enters the cluster, the GCM sends a message which indicates that all its Monitors and ScalingProcesses must be killed. This way we ensure that the nodes which join the cluster are clean.

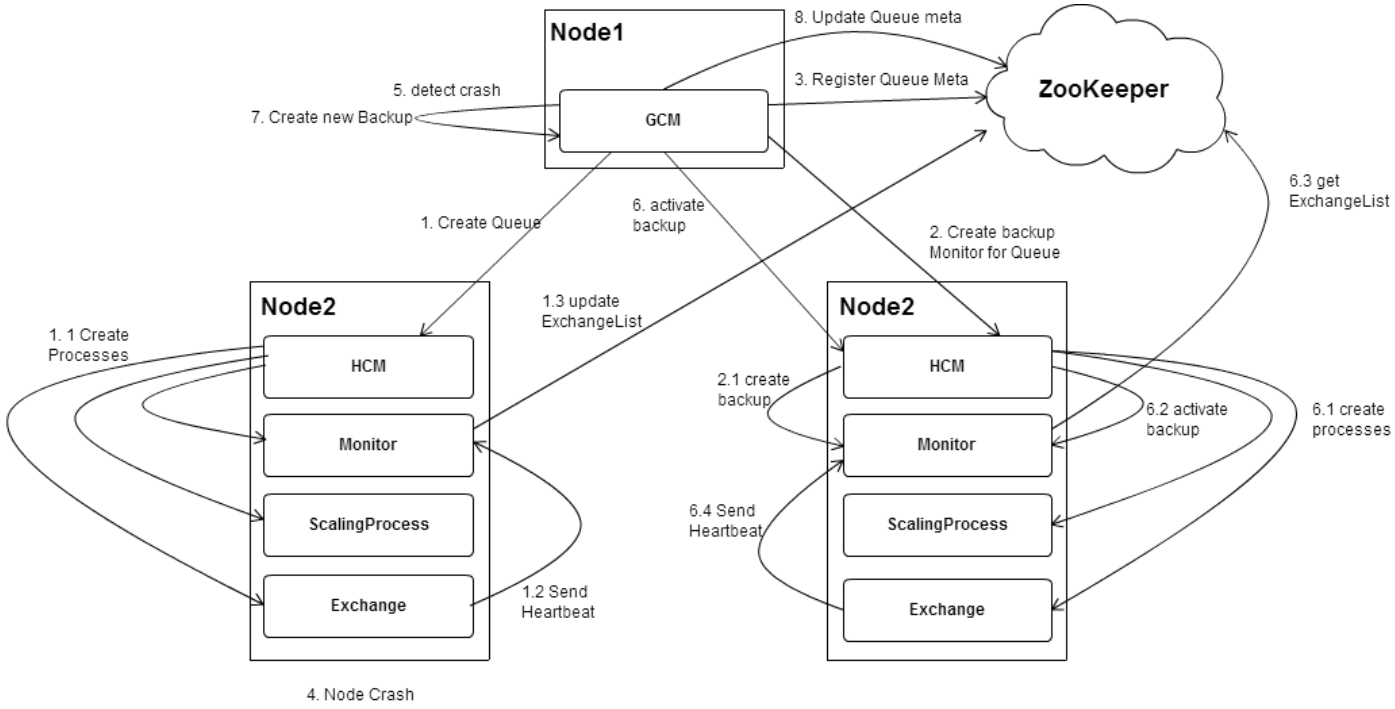

Figure 8.3: Cluster with two HCMs and one queue, HCM failover mechanism

Chapter 9

Implementation details

9.1 Introduction

In this chapter, we introduce the implementation of the solution presented in the previous one. The implementation has been split into three milestones. The first concerns the High availability for the master (GCM). The second is about the monitoring of the processes which run on a node (crash detection and recovery). The last concerns the HCM node crash detection and recovery. Because describing code can be exhaustive and not very interesting, we will describe the implementation with high level algorithms and diagrams.

9.2 GCM High Availability

9.2.1 Failure detection and recovery

In the state of the art, we introduced *Curator* (section 2.4), a client library for ZooKeeper which implements reliable recipes. It provides an implementation for *leader election* which allows coordination among processes in order to elect one of them as *leader*. In addition, the implementation provides handlers to deal with edge cases such as network partition, leader crash, etc. In this way by using this recipe, we are able to implement the solution presented in the previous chapter. A GCM starts in the following way:

1. The GCM opens a Curator session.
2. It starts the leader election.

3. It starts the ManagementController thread.

4. The GCM and its ManagementController wait for leadership (*state 0*).

At this moment, we have one or more GCM instances which wait to become the leader. Eventually one and only one of them will become the leader (guarantee provided by Curator). Once a GCM obtains the leadership, its main thread and its ManagementController start processing client requests. The main thread also registers a znode in ZooKeeper which contains the GCM's address. Now, two situations can occur, the process can crash or it can lose its session with ZooKeeper.

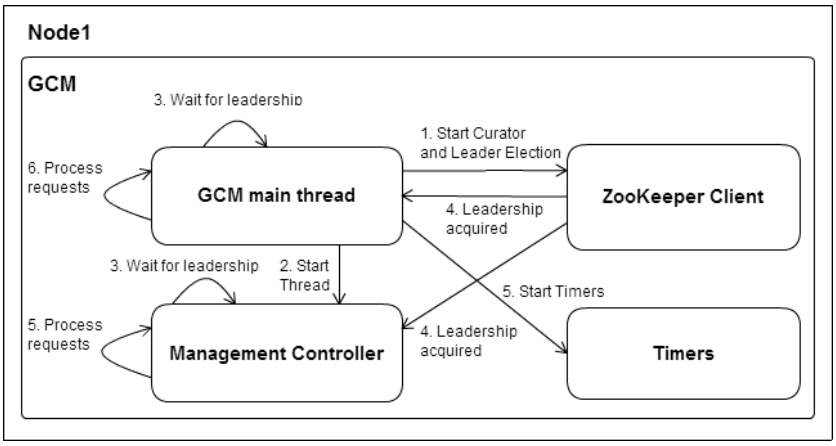

Figure 9.1: GCM leader election process

Session lost

At each instant, the GCM can lose its leadership if it loses its session with ZooKeeper, e.g. in case of a connection problem. If it loses the leadership it cannot process requests anymore. We must then ensure that GCM is in *state 0*. To reach this state, it performs the following steps:

1. After the request that they currently process has completed, the GCM and the ManagementController stop handling requests.

2. The GCM stops its Timers, meaning it stops broadcasting the cluster's topology.

3. The GCM and its ManagementController wait for leadership (*state 0*).

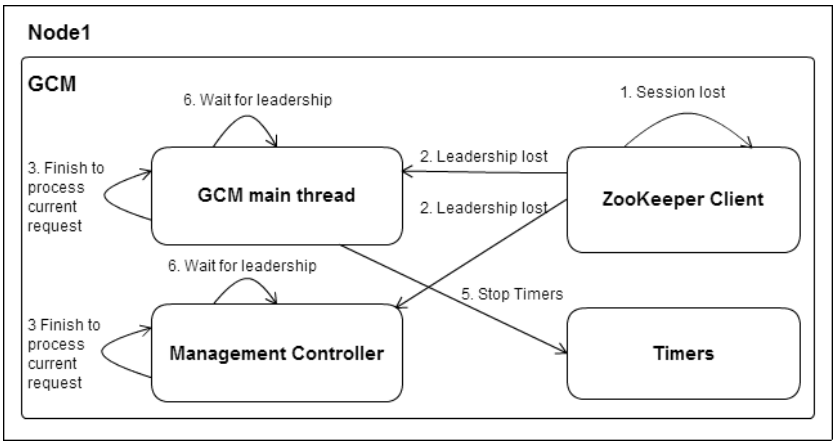

Figure 9.2: GCM ZooKeeper session lost

Node crash

A GCM is either active (leader) or passive. If a passive GCM crashes, it will stop sending heartbeats and thus lose its session with ZooKeeper. As a result it will not participate anymore in leader election. If the active GCM fails, it will also lose its session with ZooKeeper and Curator will promote a passive GCM as leader. A cluster with n GCM processes supports up to n - *1* crashes.

9.2.2 Connection establishment with leader

As we mentioned in the previous subsection, when a leader acquires leadership, it registers a node in ZooKeeper which contains its address. Rather than passing directly the GCM's address to the clients as in the original implementation, we only provide ZooKeeper's addresses to them. In this way, before establishing a connection with the leader to send their requests, clients acquire the leader's address from ZooKeeper.

When a passive GCM receives a request from a client, it responds with an error code which indicates that it is not the current leader. A client that receives this error code as a response must fetch the current leader's address from ZooKeeper.

9.2.3 Idempotent requests

Electing a leader is not sufficient to avoid zombie processes, i.e. processes which are running but are not registered in ZooKeeper. We also need to make all requests idempotent to ensure that processes are not duplicated. Indeed, in the original implementation, a GCM crash could lead to zombie processes.

, For instance, if a client sends two times a queue creation request, if the first request failed because the GCM has crashed just after if created the processes (Monitor, Exchange, etc.) but before it registered metadata, the second request will recreate the same processes. Thus, the processes will be duplicated.

To solve this issue, the requests which can lead to this kind of situation must become idempotent. An idempotent request can be send many times to the GCM without process duplication, data duplication, error, etc. We identified two requests which are not idempotent in the original implementation: *Queue creation* and *Exchange creation*. The solutions for them consist to implement a transaction mechanism.

Queue creation

When receiving a queue creation request, the GCM selects the hosts on which the Monitor and its backups will be installed. After this step, the GCM registers in ZooKeeper a transaction which contains the queue name and the addresses of these target hosts. This transaction remains in ZooKeeper until the queue creation process has completed.

Each time a queue creation process is received, the GCM checks if a transaction exists for this queue. If the transaction already exists, the GCM gets the HCM addresses from the transaction and then sends process creation requests to the corresponding HCMs. The procedures which create processes on HCMs are idempotent, in this way the processes cannot be duplicated. Finally, the GCM registers the metadata for the queue and then removes the transaction.

Exchange creation

The solution is the same as for queue creation: an HCM is selected and then a transaction is registered. For each Exchange creation request, the GCM checks if the transaction for this Exchange already exists. Each transaction

is identified by an ID provided by the client. If the transaction exists, the request to create the Exchange is resent to the same HCM. The procedure to spawn an Exchange on the HCM is idempotent. Once the Exchange is created, the transaction can be removed.

9.3 Process crash detection and recovery

In the previous chapter, we mentioned that the processes which run on a slave must be monitored by the HCM, in the same way as Storm's tasks are monitored by their Supervisor.

The implementation of the monitoring system in RoQ has been widely inspired by Storm's implementation and some of its classes have been directly imported in RoQ (as allowed by the licenses).

9.3.1 Process creation

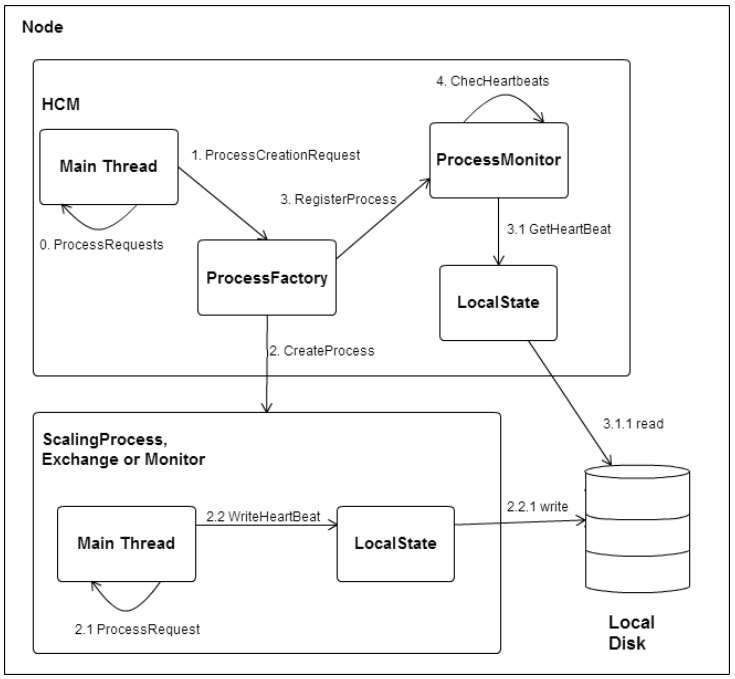

Figure 9.3: Process Start up

Figure 9.3 shows the creation procedure for one process (Exchange, ScalingProcess, Monitor). Processes are created through a system call by the ProcessFactory which also registers them in the ProcessMonitor. The ProcessMonitor is a thread which periodically checks for process heartbeats in the LocalState. The LocalState is a key-value store that was imported from Storm in which keys are process IDs (actually their frontPorts) and values

are timestamps. Processes periodically write to this store to indicate that they are alive.

9.3.2 Process recovery

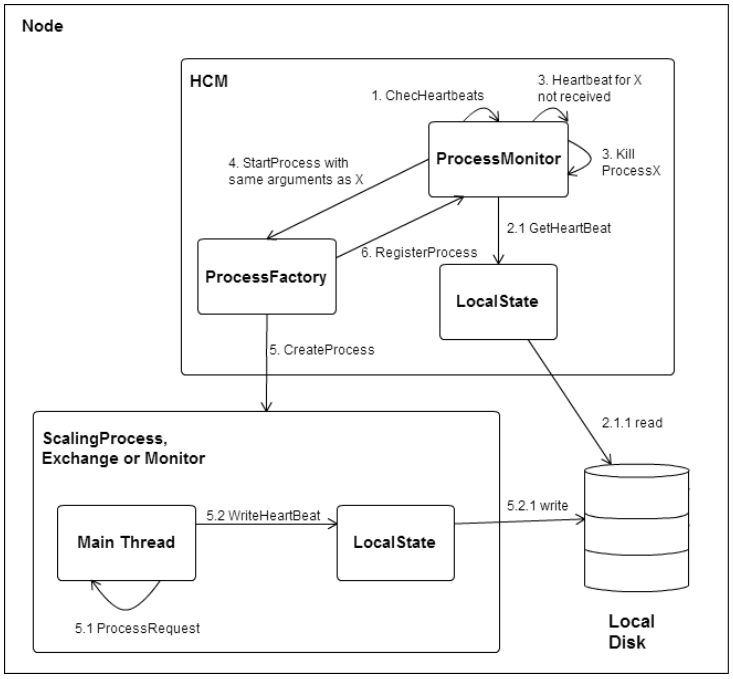

Figure 9.4: Recovery process

Figure 9.4 presents the recovery procedure for a process which has not sent its heartbeat. If the heartbeat has not been written on time by a process, the ProcessMonitor reads the LocalState, detects a missing heartbeat, then kills and restarts the offending process.

The ProcessMonitor keeps track of information about the process, such as the ports used, the queue name for which the process works, etc. These pieces of data correspond to the process' arguments used at start up.

The following subsections provide additional details about the recovery procedure for each type of process.

Monitor recovery

When a monitor starts or recovers, it gets its Exchange list from ZooKeeper. This list is stored in ZooKeeper and updated each time an Exchange leaves or joins the queue.

ScalingProcess recovery

When this process starts, it automatically sends a request to the GCM in order to receive the scaling rules. This is the only information which composes its state. In this way this step is sufficient to recover. Information such as statistics is periodically sent by the StatMonitor and the Subscribers. In addition when restarting a ScalingProcess, we bind it to the same ports, so that no special actions are needed to reestablish the connection.

Exchange recovery

Exchanges are stateless. In this way, an Exchange which replaces a lost one provides the same service and doesn't require additional recovery steps. Since the new process is bound to the same ports, there are no special actions to perform to reestablish the connection with clients.

9.4 HCM crash detection and recovery

This last section concerns the implementation of the recovery process for an HCM. As we mentioned in the high level solution (section 8.3.3), we used the Curator Service Discovery to register the HCM nodes. When one of these joins or leaves the system, a function in the *HCMListener* is triggered.

If a node has left the system, we consider that we must recover the cluster state by starting a backup Monitor for each active Monitor which was present on the dead node. To perform this action, the HCMListener sends a message to the ManagementController to indicate which HCM has left. Thanks to this information the ManagementController can find the **list of backup and active Monitors** which were present on the node that has left.

First, to recover the queues, the ManagementController does the following actions for each active Monitor in the list:

- It selects the backup Monitor that will become active.
- It looks for the HCM which handles this backup Monitor.
- It sends a message (via the QueueFactory) to the HCM to activate the backup Monitor.
- It sends a request to the GCM to update the queue's metadata.

The activation process for a backup Monitor consists of the following steps:

- It fetches the list of Exchanges in ZooKeeper.
- It sends a message to each Exchange to indicate its own address.
- It relocates the producers which are connected to dead Exchanges.
- It broadcasts the list of Exchanges to the subscribers.

Once these steps have completed, the ManagementController restores the Replication Factor for each queue. For each Monitor that was lost in the HCM crash (backup or active):

- The ManagementController finds a candidate HCM on which to create a new backup Monitor. We must take care to avoid duplicating two Monitors for the same queue on the same host.

- It creates the backup Monitor by sending a request to the HCM.
- It updates the queue's metadata by sending an update to the GCM.

Chapter 10

Tests and edge cases

10.1 Introduction

In this chapter we present the way to test the features implemented in the previous one. This phase is important to valid our implementation and to check if this one fit with the requirements to make a system Highly Available. Implementing tests for these features is not straightforward because we must simulate different scenarios which can occurs in a distributed environment. Actually, we will not describe the unit tests to stay focused on the integration tests which are more interesting to explain.

First, we will present the test framework that we developed. Second we will describe the integration tests realised with this framework. Third, we will discuss about the edge cases which have not been covered by our solution.

10.2 Test framework

10.2.1 Docker

Docker in a nutshell

Docker [36] is a platform which allows developer and sysadmin to deploy their processes in containers. A container is isolated from the rest of the host and has its own file system and virtual network interface. In this way, each of them has its own environment (such as libraries, binaries, etc.) to run its processes. The containers are handled by the Docker supervisor, a process responsible for managing them and which runs on the host.

Docker allows us to run multiple containers on the same machine. Users can communicate with them through the ports of their virtual network interface. These ports can be bound to host's ports in order to allow an application which runs inside a container to communicate with the outside world.

The main difference with a virtual machine (e.g. a machine emulate by virtualbox) is that containers don't run a complete guest OS (see figure 10.1), they share the OS of the host machine. For this reason, a container is very lightweight compare to a virtual machine. Therefore, a modest machine can run lot of them.

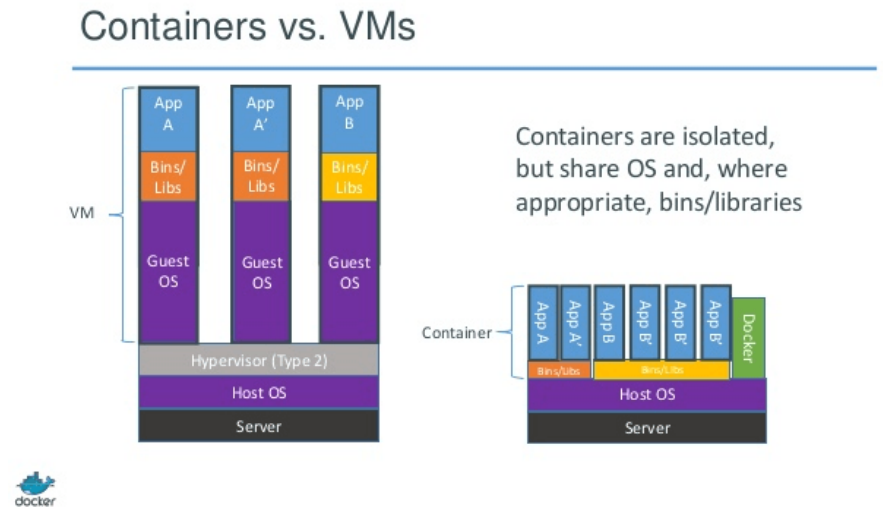

Figure 10.1: Comparison of VM and container, taken from [36]

Interest of Docker

The main reasons which led us to use Docker are the following:

- **Small footprint**, we are able to run many containers on a modest machine.
- **Virtual network interface**, each container has its own local IP address.
- **Easy to use**, a developer is able to quickly deploy a process in a container.

The second point is particularly interesting because it implies that two processes which are deployed on different containers must communicate like if they are on two different machines. In this way, we are able to deploy processes (GCM, HCM, ZooKeeper) on different containers and configure them to communicate together thanks to the IP address of their virtual network interface. As a result that they adopt the same behaviour than in a production environment.

In addition, each process is bound to its virtual interface, so we avoid the problems of ports which are already used when we deploy many processes (e.g. two GCMs, three HCMs) on the same machine.

These features are very interesting for our test framework, because they allow us to simulate a distributed environment on a single machine. We are also able to simulate some scenarios such as node crash, by just killing a container.

10.2.2 Integration of Docker with JUnit

The integration and unit tests for RoQ are currently implemented with JUnit. For this reason, we will continue to use this framework in order to test our implementation.

Actually, implementing tests for our features with JUnit only is possible but very hard. For this reason we chose to create a library which is able to provision containers. For instance, the library can start a container which contains a GCM, an HCM or a ZooKeeper process. In addition, the library configures the processes to make them able to communicate in order to form a cluster.

10.3 Integration tests realised

10.3.1 GCM

GCM failover

The test must verify if the failover mechanism works as expected. The tests consist to start a cluster with our library which contains two GCMs, one ZooKeeper node and one HCM. Thus, we have 5 containers. Once the cluster started, we create a queue in the system. Once the queue created, we kill the active GCM.

Once killed, the second GCM must become active (failover), so we try to create the same queue, the GCM must respond that the queue already exists. we try then to connect a Publisher and a Subscriber to the queue, if they successfully established the connection the test has passed.

If the test has passed, we can establish that the failover mechanism works as expected.

Idempotent requests

In the previous chapter, we discussed about the transaction mechanism to create queues and Exchanges. To test this feature, we don't need to use the Docker library. The test for queues and Exchanges follows the same steps, we describe here the general way to build this test.

First, we start a GCM, two HCM (called HCM 1 and HCM 2) and ZooKeeper on the machine. Once the components started, we put the cluster in the following state:

- A transaction for the queue (resp. Exchange) *queueX* already exists.
- the processes resulting from the queue or Exchange creation already exist on the HCM 1.
- The metadata has not been registered in ZooKeeper.

Once the cluster in this state, we test the transaction mechanism by sending a request to create the queue (resp. the Exchange) *queueX* and we pass the HCM2's address as argument. If the transaction mechanism works, the cluster must ignore this address and must take the HCM's address presents in the transaction.

If the GCM returns that the queue (resp. the Exchange) has been created, we check if the processes have not been duplicated. In this case the test has passed and we can establish that the transaction mechanism works as expected.

10.4 HCM

10.4.1 Processes crash recovery

For these tests, we don't need to use our Docker library. The goal of them is to establish if the ProcessMonitor works as expected. To perform these

tests, we create a GCM an HCM and a ZooKeeper instance on the machine. Then we decline the test, in three sub-tests.

Active Monitor crash recovery

We send a request to the HCM in order to create a queue. We kill then the Monitor. Once the Monitor dead, we create a Publisher and a Subscriber which must establish a connection with the queue. This request can takes long time because we must wait for Monitor recovery. If the Publisher and the Subscriber finally establish the connection, the test has passed because that means that the Monitor has recovered.

In addition, we must check if the Exchanges list has been recovered. For this test, we create a queue with three Exchanges, then we kill the Monitor and we waits for its recovery. Once it has recovered, we check if the Monitor has a list of three Exchanges. In this case the test has passed.

Backup Monitor crash recovery

For this test, we create a queue with one active Monitor and one backup Monitor. We kill the the backup Monitor and we wait for its recovery. Once that it recovered, we check if the recovered process is in backup mode, then we kill the HCM which handles the active Monitor (see section 8.3.3). Once the recovery process finished, the backup Monitor must be active. In this case the test has passed.

Exchange crash recovery

We send a request to the HCM in order to create a queue. Once the queue created, we create a Publisher and a Subscriber which establish a connection with this queue. The Publisher begins to send its messages and the Subscriber receives them.

We kill the Exchange on which the clients are connected. At this moment, although the Publisher continues to send messages, the Subscriber doesn't receive them. If the recovery process work, the Exchange must eventually recovers and the Subscriber must receive message again.

ScalingProcess crash recovery

To test this process, we load first a scaling policy on the GCM and then we start a queue. Second we crate Publishers (resp. Subscribers) which send (resp. receive) messages. We kill then the current ScalingProcess and we wait for its recovery.

Once, it has recovered, we checks if it continues to receive statistics from Subscribers and StatMonitor. We also check if it scaling policy corresponds to the policy defined. In this case the test has passed.

10.4.2 HCM crash recovery

This test must check if the HCM crash (and by extension node crash) is supported and if the cluster can recover from this disaster. We create a cluster (via our Docker library) with one GCM, four HCMs (HCM 1, 2, 3, 4), and one ZooKeeper. We set the Replication Factor to three.

Once the cluster started, we create two queues, *queue1* which has its active Monitor on the HCM 1 and *queue2* which has its active Monitor on the HCM 2. We create a Publisher and a Subscriber which are connected to the *queue1*. The publisher sends its messages and the Subscriber receives them.

We kill the HCM 1. The Subscriber stops to receive message and the GCM must detect that a node has been lost, then the GCM starts the recovery process. At the end of the recovery process, the subscriber must receive messages again. In addition, the Replication Factor for the queues must be maintained.

If the Replication Factor for *queue1* has been maintained, we can establish that the active Monitor lost has been replaced and that an additional backup has been created. If the Replication Factor for *queue2* has been maintained, we can establish that the backup monitor lost has been replaced by another one.

If all these condition has been checked, the test has passed.

10.5 Edge cases

In this section we present some features which must be implemented to completely finish the implementation. Basically, all the most important features

have been successfully implemented and the described tests pass. But some improvements can be brought to make the system ready for production.

10.5.1 Maintaining the connection with every Monitors

When it establishes a connection with a queue, the Subscriber receives the Monitors list (backups and active) and it receives messages from them. When new backup Monitors have been created, the Subscriber must update its Monitors list in order to establish a connection with this one. This feature (update Monitor list) must be implemented.

10.5.2 Daemon process to Monitor the HCM

We described in the high level solution (section 8.3.1) a simple process which is responsible for Monitoring and restarting the HCM if this one fails. This improvement allows to reduce the *mean time to recovery* by avoiding to trigger a complete node recovery when the HCM process has crashed.

Implementing this feature is not straightforward, because the HCM maintains a complex state which must be restored when it is recovered.

10.5.3 Maintaining Replication Factor in any situation

Currently, it is possible that the Replication Factor is not maintained in different scenarios:

- There is no nodes enough. If we have two HCMs and that the Replication Factor is set to three, when we create a queue, this one has two replicas. We must implement a way to create an additional replica when an HCM joins the cluster.
- The backup creation request failed. When the QueueFactory sends a request to the HCM in order to create a backup Monitor, we suppose that the request will not fail. This hypothesis is not acceptable, we must implement a way to create the backup on another HCM if the first one failed to create it.

10.5.4 Simultaneous crash of GCM and HCM

For the moment, we suppose that an HCM and the active GCM don't crash at the same time. Although this situation is not very common, it would be interesting to handle this edge case. Currently, the problem is the following:

- An HCM crashes.
- The GCM detects the crash and fails during the recovery process.
- The GCM which becomes active doesn't continue the recovery process.

The solution for this problem is not straightforward and consists to implement a transaction mechanism like for queue/Exchange creation process. This transaction must indicate the node which crashed. When a GCM is elected, it checks first if there exist a transaction to process and it processes it. The recovery process must become idempotent to ensure that the cluster state stays valid.

Part IV

Conclusions

Chapter 11

Future works

RoQ is now a highly available messaging service which can deliver messages with a high throughput. However, high availability is only the first step toward providing a competitive system.

To continue the evolution of RoQ, the next step would be to solve the edge cases with the solutions proposed in section 10.5. We could then implement the *at least once* semantic in order to provide a stronger delivery guarantee.

Indeed, RoQ can lose messages due to Exchange failures: if an Exchange fails, we lose its buffered messages and the recovery mechanism is not able to restore them. In section 7.3.2, we proposed an abstract solution to this problem that consists in mirroring messages across Exchanges in order to ensure they are persisted. RabbitMQ's solution matches our architecture (as opposed to Kafka), and implementing it in RoQ would be possible.

Chapter 12

Final words

This master thesis about high availability in RoQ has taught us ways to build a fault tolerant system which aim to minimise downtime in order to improve availability. We saw that making a system highly available depends more on its architecture than on the service it provides. The state of the art presented different cutting edge technologies which are not always related to messaging systems. Some of them have architectures close to RoQ's. The solution we chose for RoQ was inspired by HBase and Storm, two mature technologies which propose solutions for high availability in a master/slave architecture. In addition to this, the state of the art presented solutions to particular problems for different types of distributed systems. For instance the computation semantic in Storm, the delivery semantic in RabbitMQ and Kafka, the consistency in HBase and Cassandra. Some of these solutions from RabbitMQ and Kafka are interesting for future works on RoQ as we mentioned in the previous section.

Part II showed us how to reuse a solution from an existing system and adapt it for our own purposes. The approach consisted in describing the current architecture of RoQ in order to determine the similarities with the ones described in the state of the art. We then looked at all the places where RoQ's architecture prevented it from being highly available. With those problem areas identified, we were able to propose an abstract solution based on the state of the art. Lastly, after splitting the solution into separate steps, we implemented it as described in chapters 8 and 9.

Finally, we presented an original way of testing failure scenarios for a distributed system. To achieve this, we build a framework which uses a Docker library in order to simulate a cluster on the developer's machine.

Since Docker is lightweight, this way of testing allowed us to efficiently debug our features.

Bibliography

[1] W. Emmerich, "Distributed system principles." http://www0.cs.ucl.ac.uk/staff/ucacwxe/lectures/ds98-99/dsee3.pdf, 2007.

[2] "Wikipedia: Peer-to-peer." http://en.wikipedia.org/wiki/Peer-to-peer. [Online; accessed 22-May-2015].

[3] "Wikipedia: Overlay network." http://en.wikipedia.org/wiki/Overlay_network. [Online; accessed 28-May-2015].

[4] N. A. L. Seth Gilbert, "Perspectives on the cap theorem," 2012.

[5] P. V. Roy, "Lsinf2345: Languages and algorithms for distributed applications," 2014.

[6] R. G. A.-M. K. Patrick Th. Eugster, Pascal A. Felber, "The many faces of publish/subscribe," 2003.

[7] S. R. Christian Esposito, Domenico Cotroneo, "On reliability in publish/subscribe services," 2013.

[8] F. P. J. Patrick Hunt, Mahadev Konar and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems,"

[9] "Apache curator." http://curator.apache.org. [Online; accessed 10-Maart-2015].

[10] "Apache zookeeper." https://zookeeper.apache.org. [Online; accessed 10-Maart-2015].

[11] B. C. R. Flavio P. Junqueira and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," 2009.

[12] P. M. Avinash Lakshman, "Cassandra: A decentralized structured storage system." http://cassandra.apache.org. [Online; accessed 12-October-2014].

[13] D. L.-N. D. R. K. M. F. K. F. D. H. B. Ion Stoica, Robert Morris, "Chord: A scalable peer-to-peer lookup protocol for internet applications," 2001.

[14] A. Lakshman and P. Malik, "Cassandra: a decentralized structure storage,"

[15] R. Y. N. Hayashibara, X. Defago and T. Katayama, "The ϕ accrual failure detector," 2004.

[16] D. Borthakur, "Hdfs architecture guide." http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. [Online; accessed 20-October-2014].

[17] "Hdfs high availability using the quorum journal manager." http://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html. [Online; accessed 20-October-2014].

[18] L. George, "Hbase architecture 101 - write-ahead-log." http://www.larsgeorge.com/2010/01/hbase-architecture-101-write-ahead-log.html. [Online; accessed 12-October-2014].

[19] "Hbase documentation." http://hbase.apache.org/book.html. [Online; accessed 12-October-2014].

[20] D. D. Enis Soztutar, "conference: Hbase read high availability using timeline-consistent region replicas." http://fr.slideshare.net/enissoz/hbase-high-availability-for-reads-with-time. [Online; accessed 19-October-2014].

[21] A. S.-K. R. J. M. P. S. K. J. J. K. G. M. F. J. D. N. B. S. M. D. R. Ankit Toshniwal, Siddarth Taneja, "Storm @twitter," 2014.

[22] "Executors." https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html. [Online; accessed 21-December-2014].

[23] "Apache thrift." https://thrift.apache.org/docs/. [Online; accessed 21-December-2014].

[24] "Storm documentation." https://storm.apache.org/documentation/Home.html. [Online; accessed 20-December-2014].

[25] J. R. Jay Kreps, Neha Narkhede, "Kafka: a distributed messaging system for log processing,"

[26] J. Rao, "Intra-cluster replication in apache kafka." https://engineering.linkedin.com/kafka/intra-cluster-replication-apache-kafka, 2013.

[27] N. Narkhede, "Kafka replication."

[28] "Rabbitmq: What can rabbitmq do for you?," [Online; accessed 5-April-2015].

[29] "Amqp advanced message queuing protocol protocol specification (v0.9.1)," 2008.

[30] "Rabbitmq: Amqp 0-9-1 model explained," [Online; accessed 5-April-2015].

[31] "Rabbitmq: Highly available queues." https://www.rabbitmq.com/ha.html. [Online; accessed 5-April-2015].

[32] S. MacMullen, "Guaranteed multicast." http://hg.rabbitmq.com/rabbitmq-server/file/bb9b95480101/src/gm.erl. [Online; accessed 22-April-2015].

[33] E. Z. Nam Luc Tran, Sabri Skhiri, "Eqs: an elastic and scalable message queue for the cloud," 2011.

[34] A. D'erman, "Roq architecture," 2013.

[35] P. Hintjens, "Ømq - the guide." http://zguide.zeromq.org/page:all. [Online; accessed 12-December-2014].

[36] "Docker documentation." https://docs.docker.com. [Online; accessed 28-May-2015].

[37] M. J. F. S. S. I. S. Matei Zaharia, Mosharaf Chowdhury, "Spark: Cluster computing with working sets," 2010.

[38] "Spark documentation." https://spark.apache.org/docs/latest. [Online; accessed 10-Maart-2015].

[39] S. Ryza, "Apache spark resource management and yarn app models." http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/, 2014.

[40] T. D. A. D. J. M. M. M. M. J. F. S. S. I. S. Matei Zaharia, Mosharaf Chowdhury, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," 2012.