Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Adapting a noSQL database for autoscaling on a PaaS environment

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Fraseniuc, Bogdan-Alexandru

Tutor: Bernabeu Aubán, José Manuel

2015-2016

Resumen

Este trabajo se afana en crear un servicio que ofrece la funcionalidad de un clúster fragmentado MongoDB y que al mismo tiempo es capaz de auto escalar dentro de una plataforma como servicio.

La base de datos utilizada es MongoDB y la plataforma es Kumori PaaS.

Esta solución tiene la ventaja de que permite al desarrollador de software centrarse más en la parte de desarrollo del servicio que en la gestión de la base de datos.

Para alcanzar este objetivo se realiza un estudio sobre los mecanismos internos de MongoDB, se proporciona una vista general sobre cómo funciona Kumori PaaS y se crean dos aplicaciones de servicio. Estas aplicaciones de servicio tienen topologías distintas: una más simple que no soporta auto escalado pero que se usa para ver si MongoDB podría funcionar sobre esta plataforma, y una más compleja que se parece a la topología de un clúster fragmentado MongoDB y que además ofrece auto escalado.

Palabras clave: Computación en la Nube, Bases de datos, NoSQL, autoescalado, PaaS.

Abstract

This project strives to create a service offering the functionality of a MongoDB sharded cluster while also being able to autoscale on a Platform-as-a-Service.

The database used is MongoDB and the platform is the Kumori PaaS.

The benefit of this solution is that it allows a software developer to focus more on the development of the service instead of the management of the database.

In order to achieve this objective a study of the MongoDB internal mechanisms is done, an overall view of how the Kumori PaaS works is given and two service applications are created. These service applications have different topologies: a simple one which does not support autoscalability but is used to view if MongoDB can work on this platform, and a more complex one which resembles the topology of a MongoDB sharded cluster and also offers autoscalability.

Keywords : Cloud Computing, Databases, NoSQL, autoscaling, PaaS.

Table of contents

Table of figures

1. Introduction

1.1 Scope

In this day and age, everything seems to be moving to the cloud, and this includes databases. One potential solution for someone wanting to deploy a database on the cloud is using a Platform as a System (PaaS).

A PaaS is a category of cloud computing services which allows developers to create and test their applications without having to worry about the underlying infrastructure. This includes not having to worry about the network, the operating system, the storage, runtime environment or the database.

The developer wants to spend time writing the code for the actual application or service; not work on the infrastructure, concern himself with backups, scalability or how to recover from crashes. This is what PaaS tries to offer, cover the entire lifecycle of the programmer's application, allowing him to do what a developer should do, code.

The work done in this project was realized during an internship at the Instituto Tecnológico de Informática (ITI) of the Universidad Politécnica de Valencia (UPV), working alongside the team who is currently developing this PaaS.

1.2 Objectives

There are three main objectives defined: understand how MongoDB works internally, understand how Kumori PaaS works in order to build components and service applications for it, and create a service application serving a MongoDB sharded cluster which autoscales on the Kumori PaaS.

The first objective two objectives are straightforward, before trying to adapt a technology and mold it to some specific needs, a good understanding of that technology is needed. The concepts explained about MongoDB are related to a database administrator's point of view, since the main concern is how the database works internally and how to manage it, not how to code a user application which uses this database to store its data.

The Kumori PaaS, as any other PaaS solution on the market, has to offer a database solution, be it relational (SQL) or non-relational (noSQL). In order to adapt MongoDB to work inside this PaaS and be able to autoscale, it is imperative to understand how components and services are *modelled* in this platform.

The final objective is to build a service application which can offer the MongoDB sharded cluster functionality and more, autoscaling.

1.2 Document structure

This document is structured in six chapters and the content of each chapter is detailed below.

In the first chapter a brief introduction to the project is made and the reader is presented with the scope of this project followed by the objectives which it aims to achieve.

The second chapter offers a brief explanation of MongoDB and some of its key elements and concepts, the objective of this chapter is to explain how replication and sharding is achieved.

The third chapter offers insight on how the actual platform on which the MongoDB service application will be deployed, works.

The fourth chapter describes a first approach to the design and implementation of the service.

The fifth chapter presents the topology needed to achieve the objective proposed at the beginning of the project: adapt a noSQL database for autoscaling on a PaaS environment.

The sixth chapter goes into detail about the conclusions and future work that still has to be done.

There are two appendices:

- Appendix A: more details about manifests
- Appendix B: certain parts of the source code used and scripts

2. MongoDB

2.1 Introduction

MongoDB is a noSQL database. The word “noSQL” has had different meanings over the years: “non SQL”, “non relational” or even “not only SQL”. This term is used to refer to database technologies which try to adapt to some of the needs (high availability and horizontal scalability) of Web 2.0 [1] companies such as Google and Amazon. [2]

Essentially, MongoDB is an open-source document database, which implies that instead of storing the data in rows (as in traditional relational databases), data is stored inside documents. It can be said that the equivalent to SQL databases rows and tables are represented inside MongoDB by documents and collections, and the main difference between these two concepts is that in SQL a strict schema has to be defined, whereas in MongoDB a dynamic schema is used. [3]

At the moment, it is the most used noSQL database solution on the market and it has even surpassed PostgreSQL in popularity. MongoDB is fourth on the overall complete database ranking; with Oracle, MySQL and Microsoft SQL Server being the top three solutions [4]. It should be noted that the top three on this ranking list, all represent relational database models, and these technologies have been worked on and improved over the last 40 years.

One of the reasons why it got so popular is because of the design philosophy behind its architecture, the so called MongoDB Nexus Architecture [5]. The objective is to avoid “reinventing the wheel”, there are many features in relational databases which are critical (e.g. secondary indexes and consistency), so the idea is to maintain this foundation while adding the innovations of noSQL (e.g. flexible data model and scalability).

In conclusion, apart from being a noSQL document database, it also strives to provide some other key features of relational databases. The goal is to achieve a mixture between noSQL and relational technologies in order to satisfy the requirements of modern applications.

2.2 MongoDB Internals

In order to adapt MongoDB and integrate it in a PaaS, a better understanding of some key features of the ecosystem is needed. As mentioned in the introduction of this chapter, noSQL databases try to address the issues of high availability and horizontal scalability. According to the MongoDB Nexus Architecture, which can be seen in the Figure 2 below, innovations offered by noSQL systems would solve these issues [5].

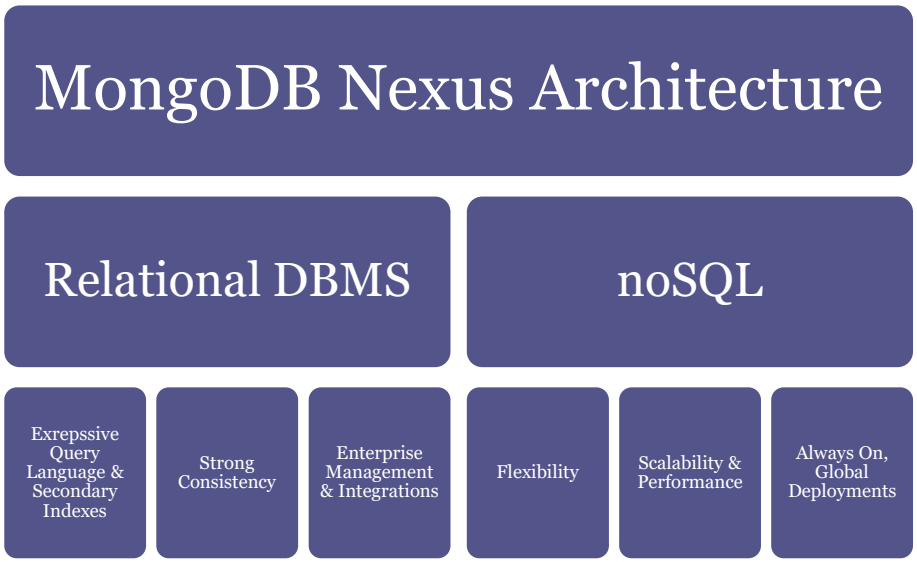

***Figure 1**: MongoDB Nexus Architecture*

Before moving forward and explaining the mechanisms which MongoDB uses to provide high availability and horizontal scalability, it is useful to explain these two concepts.

High availability basically means that the system will be available to its users for a higher than normal period of time, which has been agreed upon beforehand. In the case of MongoDB it relates more to data availability, since this is the service that a database provides. The main thing to take into consideration is not to confuse the uptime of a system with the availability of the service provided by it. A given server could be up and running, fully configured with a correct installation and deployment of a service but because of different possible problems, like a network outage for example, the service offered would not be available to the users.

The solution adopted by MongoDB to provide high availability is replication, which implies having the service running on multiple nodes instead of just one. All data is copied to all the nodes, this way data redundancy is provided, and if one node fails, another one can serve the data to the users. It also provides automatic failover, meaning that in the case of a node failing, the whole process which setups and allows a different node to offer the service, is done by the system itself without intervention of a database administrator. This way, the availability time of the service is increased even more because it is not necessary to wait for someone to detect the problem and fix it afterwards.

The other issue which remains to be addressed is horizontal scalability. First of all, the concept of scalability refers to the process of adding more resources to a system in order to deal with an increasing workload [6]. There are two solutions available, scaling up (vertical scalability), which resolves the problem by upgrading the hardware of the node (e.g. more ram, better cpu); but this is limited by the available hardware on the market and there is an obvious top limit which can be achieved, a certain point beyond which the node cannot be improved anymore. This approach is also quite expensive

given that the hardware must always be changed with newer one and also because the nodes tend to be high end servers.

The alternative to vertical scalability is scaling out (horizontal scalability). As opposed to the first solution, instead of using top of the line servers, multiple commodity systems are used. So instead of improving the hardware of the node, whenever more resources are needed, more nodes are added. This approach is cheaper because the nodes are cheaper. But the same thing that makes this solution better (because it is cheaper and easier to attain) also makes it worse, given the fact that commodity servers have more hardware failures than servers. This means that the software now has to handle with individual nodes failing.

In order to achieve horizontal scalability, apart from using replication, MongoDB also uses sharding. A brief explanation of replication was given earlier, but both of these concepts will be further explained in the next subsections of this chapter.

2.2.1 Replication

Most of the explanations from here on out and until the end of this chapter are based on the MongoDB manual [3] and the author's own expertise on the matter. This expertise was gained by using MongoDB for a period of half a year, during which different deployment architectures of the service have been tested in order to better understand how this database operates.

As previously mentioned, one of the needs that noSQL databases try to fulfil is high availability, maximize the availability of a service and set in place different mechanisms which can deal with potential failures of the system in such a way that the user will not be affected. The mechanisms used to solve this issue are automatic failover and data redundancy, which are included in the replication feature of this database technology.

Before explaining what replica sets are and how they behave, a brief explanation of the core MongoDB processes and different components is necessary.

The MongoDB core processes are:

- *mongod*, which actually is a daemon [7] process (hinted by the fact that the name of the process ends with the letter "d"). A daemon process is a background process. This is the primary process for MongoDB and it handles all the data and background management operations.
- *mongo*, which is an interactive MongoDB shell. It uses JavaScript and it has two main functions: provide an interface for the system administrators and allow the developers to conduct tests directly on the database.

The different components which are part of a replica set:

- *primary*, a *mongod* process which handles are the write requests; there cannot be more than one *primary* in the same replica set at the same time.

- *secondary*, a *mongod* process which holds a copy of the data on the *primary* and which can become a *primary*.
- *arbiter*, a mongod process with no data, which is added to the replica set in order to avoid an even number of members. An even number of members inside a replica set can result in a tied election for a new *primary*.

The minimum setup recommended for a replica set is two data bearing nodes and an *arbiter*, or three data bearing nodes. For the remainder of this chapter the term "replica set" will be used to refer to a setup made up by one *primary* and two *secondary* members. This is one of the most common setups but a replica set can have up to fifty members in total, but only seven members can vote.

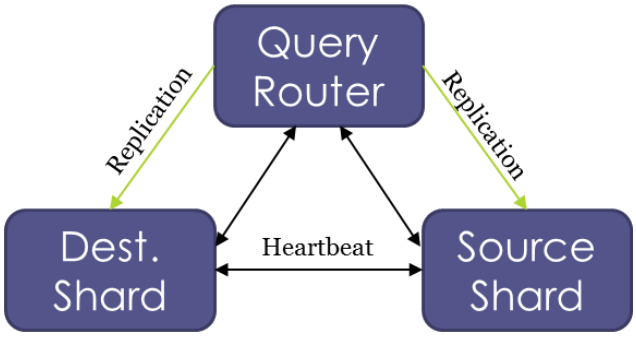

***Figure 2**: MongoDB three member replica set*

The *primary* member is the node which accepts all reads and writes by default. After applying the database operations on the *primary*, MongoDB records these operations on the *primary's* oplog. The oplog is a special collection in MongoDB where all the database operations which modify data are registered. This oplog is replicated by the *secondary* which applies the operations from the oplog to its own data set in order to maintain an identical data set throughout the replica set, and thus achieving data redundancy.

The process described above is the data synchronization method known as replication. But how do new nodes, or nodes recovering from failure, replicate the data? This is achieved by means of another data synchronization method called initial sync, which basically consists of copying all the data from one member of the replica set to another.

In order to achieve high availability one more requisite is needed apart from data redundancy, and that is automatic failover. This is the process which allows a *secondary* member of a replica set to become a *primary*. This is necessary for situations like a network partitions between the *primary* and the remaining members of the set, or when the *primary* fails because of a power outage or hardware failure. In these situations the remaining *secondary* members must be able to hold an election and chose another *primary* without manual intervention, so that the replica set can

function normally and accept writes. If no *primary* can be elected, the replica set cannot accept writes and all its members are read-only.

An immediate consequence of failovers are rollbacks (return the database to a previous state). This occurs when a *primary* accepts writes operations and it steps down before the *secondary* members have a chance to replicate these writes. The rollback is necessary in order to maintain consistency.

In MongoDB, by default, a write operations is acknowledged after it has been propagated to the *primary*. This is known as write concern *w: 1*. Starting with version 3.2, a new write concern is introduced: majority. If this write concern is used and if journaling is enabled (by default journaling is enabled) then a write will only be acknowledged after the writes have been propagated to the majority of the voting members in a replica set (this includes the *primary*) and have also been written to the on-disk journal. This journal and the oplog are different files. The oplog is used for the replication process, the journal is used for the recovery process in case of failover.

Using write concern majority has both drawbacks and benefits. The benefit explained above is the ability to avoid rollbacks in some cases of failover and to ensure data durability. But this comes at the cost of losing performance in order to assure that the write operation is truly committed. This is a perfect example to illustrate how the use case of each user is what dictates the configuration needed .If the data is sensitive and performance is not as important, then using write concern majority is the solution. In this case the system will not be available as much because of the replication lag which leads to higher latency.

An important aspect to take into account is that write concern only blocks write operations. Which means that users can read writes from the *primary* which have not been committed to the majority of replica set members. This leads to reads known as *dirty reads*, because a rollback is possible if failure occurs.

To avoid this *dirty reads*, starting with version 3.2, read concern is implemented. Two modes are available:

- local, the latest locally committed data is returned to the user. If reading from the *primary* this means that the user might read data which can be eventually rolled back. The user can see results of write operations before they are acknowledged.
- majority, only the data which has been committed to a majority of nodes is returned to the user. This fixes the issue with *dirty reads* but introduces a new problem: eventual consistency and latency. The user might be reading slightly out of date data.

2.2.2 Sharding

In the previous subsection it has been explained how MongoDB achieves high availability by using replica sets. But what happens when the working set does not fit in the RAM, or the user application generates too many queries/operations. This is when horizontal scalability becomes relevant.

In MongoDB, horizontal scalability is achieved by using sharding [8], [9]. The process of sharding is based on a simple idea: divide the data set into parts and put these parts on different servers.

In order for sharding to work, another MongoDB core process is needed, apart from *mongod* and *mongo*. This third core process is called *mongos* and its main functionality is to route queries and write operations. It is the bridge that links the user applications to the sharded cluster, it presents users with a single view of the sharded database.

By using replication, different members of a replica set have the same documents, the same data. A sharded cluster represents a set of shards, each shard being a replica set. Since the data is distributed among the different shards, this means that at any given point in time a document lives on only one shard.

In replica sets, each replica set contains a *primary* and various *secondary* members, and all the members share the same data. When using a sharded cluster, each shard (replica set) contains only a part of the data. In order for the user to access the data, the *mongos* process must be used. This process is hosted on a new component called *query router*.

The main issue now is on which criteria are the documents distributed to different shards, which steps are taken to split a collection and distribute its documents. To achieve this goal, each document in the same collection must have a unique indexed field. This will be the shard key. Each shard will host the documents which belong to a certain range of shard key values.

This concept is explained easier by means of an example. Consider a collection made up of documents which all have a common unique field *name*. This field will be indexed and chosen as the shard key. The resulting chunks and their associated shard key ranges are considered metadata and can be viewed in the Figure 3 below.

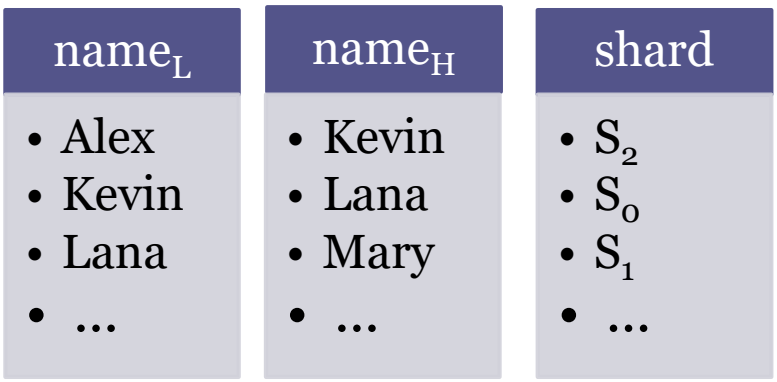

***Figure 3**: MongoDB metadata*

The Figure 3 illustrates a map that describes how the documents are distributed and to which shard they belong; a chunk is represented by a shard key range which is contained in between an inclusive lower boundary, $name_L$, and an exclusive upper boundary, $name_H$. All the documents with values of the *name* field contained in this chunk are physically located on the shard S_2.

This metadata information is not hosted on the shards and neither on the *query router*. The *query router* only keeps a cached copy of this information. The metadata for the cluster is actually located on a component called *config server*, which runs a *mongod* process and is deployed as a replica set. Apart from the metadata, the *config server* also stores other settings for the cluster, such as authentication configuration. If this component loses its *primary*, the cluster metadata becomes read-only, unless a new *primary* is elected. If the cluster metadata is read-only the system can still attend read and write operations, but no new chunk migrations and chunk splits can be realized. Without the metadata the cluster can reach an inoperable state.

2.2.2.1 Sharding – Chunk Splitting

This operation is initiated by the *query router*, it is the action which does the actual partition of the data. There is no real change to the data, it is a remapping, so it only affects the metadata. A sharded cluster can have more than one *query router*, they do not talk to each other and each one tracks writes to the chunks. This process is automatic:

1. When the data written reaches 20% of the defined max chunk size (default size is 64MB) the *query router* sends the *splitVector* command to the *primary* of the shard that owns the chunk.
2. The *primary* checks if the chunk can be split and calculates a list split points.
3. The *primary* returns a list of split points, it can be an empty list, to the *query router*.

4. If the list is not empty, the *query router* updates the *config server* in order for the metadata to reflect the splits.
5. No data changed/moved.
6. If the split points list is empty the *query router* waits until 40% and it send the *splitVector* command again.

The following Figure 4 represents the chunk splitting process.

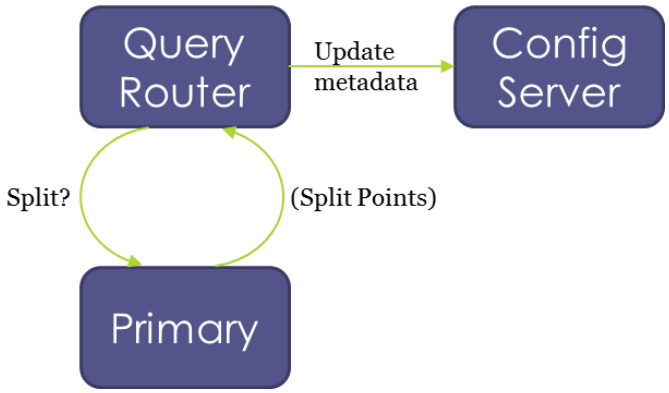

***Figure 4**: Chunk Splitting*

Manual splitting is also possible. By using the *mongo* shell, a connection can be established with a *query router* and the following commands can be used to split chunks manually:

- *splitFind("database.collection", { "field": "value" })* – splits the chunk containing the first document which fulfills the query into two chunks of the same size.
- *splitAt("database.collection", { "field": "value" })* – the document which fulfills the query will be the lower bound of the new chunk.

These commands are needed in order to deal with so called jumbo chunks; these chunks cannot be split automatically because they exceed the maximum chunk size or the number of documents contained in the chunk exceeds the maximum allowed.

2.2.2.1 Sharding – Chunk Migration

This process represents the actual movement of data from one shard to another. Different stages of the chunk migration are coordinated by the *query router* but the actual work is done by the *primaries*. The *query router* checks the config database on the *config server* and requests a balancing round in order to make sure that it can acquire the necessary locks and start the balancer. After acquiring the lock, the balancer proceeds to identify the imbalance (draining shards have more priority: shards which are being removed from the sharded cluster) and pick the chunk which has to be migrated and the migration procedure starts:

1. The balancer send the *moveChunk* command to the source shard.
2. The source shard performs sanity checks: check if the range for the chunk is valid, if the command is valid, if it is not already doing too many deletes from previous migrations.
3. The source shard initiates the transfer with an internal *moveChunk* command.
4. The destination shard checks if it has all the necessary indexes (if not, it will have to create them).
5. The destination checks for existing documents which belong in this range to avoid unique key violations, duplicate records.
6. After these checks the actual transfer begins.
7. Catch on subsequent operations. During the migration process the chunk still lives on the source shard, so all write operations to this chunk which occur during migration are directed to the source shard.
8. When the destination shard finishes catching up to all the changes which occurred during the migration, the transfer itself finishes.
9. The process now enters the critical section, the source shard connects to the *config server* in order to update the cluster metadata.
10. Once the metadata is up to date and there are no open cursors on the chunk, the cleanup starts: the source shard deletes the data which has just finished migrating.
11. Subsequent requests to the source shard asking for the migrated chunk will trigger a "stale config exception". This triggers a refresh of the metadata for the *query router* which requested the chunk, and after refreshing it proceeds to retry using the updated information.

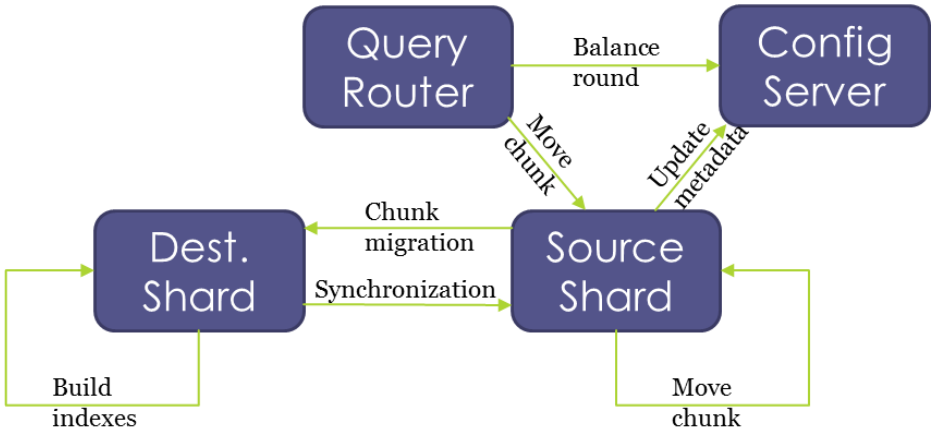

***Figure 5**: Chunk Migration*

The *Dest. Shard* which is represented in Figure 5 is the destination shard which receives the migrating chunks.

As mentioned in the chunk migration procedure, the *query router* is the component which can start a balancing round. These balancing rounds allow the

balancer to redistribute the chunks in order to achieve an even number of chunks for the collection across the shards.

The balancer is a MongoDB background process, enabled by default, which only cares about the number of chunks. It does not consider the size of the chunks or the number of documents inside a chunk because the auto-split process takes care of these problems.

The balancer activates a migration process based on the migration threshold, which is different depending on the total number of chunks for a sharded collection. These migration thresholds represent the imbalance in number of chunks between the "biggest shard", highest number of chunks for the collection, and the "smallest shard", the fewest number of chunks for the same collection. To reduce this imbalance, the chunk with the lowest range is moved from the shard with the highest count of chunks to the shard with the lowest count of chunks.

One of the possible problems induced by the automatic chunk migration are empty chunks. For example having a total of ten chunks for a sharded collection, half of them are empty and reside on one shard and the other half are not empty and reside on another shard. From the balancer's point of view, there is no imbalance, since it only checks the number of chunks.

Empty chunks can be the result of a pre-splitting mistake, or the use of a time based shard key: periodically deleting old data. The solution is to merge chunks but with a few conditions: the chunks must be on the same shard, contiguous and at least one of the chunks has to be empty.

If a chunk is empty an imbalance is created within the sharded cluster, but if a chunk is too big, this also leads to data imbalance. This can occur because of a poor shard key selection or maybe a pre-splitting error. If a chunk does not receive any traffic, the auto-split will not occur, so the series of actions to take in this situation are:

1. Do manual splits and then wait for the balancer to do its job.
2. Turn of the balancer, do manual splits and then manual migration of the chunks.

All the concepts which have been explained throughout chapter 2 give more insight on the mechanics behind MongoDB. This will be useful later on when modelling and creating the MongoDB service for the Kumori PaaS.

3. Kumori PaaS

3.1 Introduction

After explaining what MongoDB is, going over some of the more relevant features (replication and sharding) and comprehending the key components and processes which make this features possible, there is a need for a better understanding of the system for which MongoDB is adapted.

The Kumori PaaS is a Platform-as-a-Service provided by Kumory systems, a startup of the Polytechnic University of Valencia (Spain). From now on the Kumori PaaS will be referred to as simply ECloud.

ECloud is designed to manage the life cycle events of services deployed on it. The main objective is to allow the service provider to work on the development of the service instead of dealing with the service management tasks.

The current stage of ECloud is ongoing development, some of the functionality might change/evolve; only certain aspects of ECloud will be covered since going into too much detail is beyond the scope of this work. All elements defined as *to be disclosed* (TBD) are not explained, but they are shown in the example/schemas in order for the reader to get a broader view of the entire system; they are needed for the system to work correctly, but they do not affect directly (at the point in time when the work was done) the design/implementation which the author has presented. Some of these elements are still not well defined and merely informative.

3.2 ECloud service model and components

In order for a service application to work on ECloud, it must comply with the architectural patterns known as ECloud service model. According to this service model, a service application is represented by a set of interconnected components. Each of these components has a different role inside the service. The components contain the source code and a description of how to connect them to other components. The service applications contain a description of how its components are connected and also how they can be connected to other service applications.

To make possible this model, these elements are available in ECloud: component, role, channel, channel protocol, configuration resources, configuration parameters, types, runtime, service application, channel connector, role instance, service and topology.

Some of the elements must have a unique name to identify them, this is achieved by using a uniform resource identifier (URI). The structure of a URI is the following:

slap://<domain>/<elementtype>/<name>/<version>

The <domain> must be a domain name belonging to the author of the component, the <elementtype> represents the type of element being identified, the <name> is the generic name of the element within its domain and the <version> is used to identify which particular version of the element is being named. The following is an example of how a URI would look like:

slap://afraseniuc/components/mongodbsimpleproxylb/0_0_4

To define an element, javascript object notation (json) files referred to as manifests are used. An example of how these manifest are structured:

```
{
  "spec":"slap://slapdomain/manifests/service/0_0_1",
  "name":"slap://afraseniuc/services/mongodbsimpleproxylbservice/0_0_4",
...
}
```

From here on, in order to maintain a clear structure, all the example which are necessary to explain future manifest definitions can be found in the Appendix.

The *spec* refers to the specification version to be found in the manifest and the *name* is the ID of the element being defined.

As it was stated previously, service applications are made up of interconnected components. A component is viewed as an autonomously runnable executable. Even though component instances are isolated one from each other, they sometimes need functionality offered by other components. This is what will be referred to as the *dependency set*. Whereas the functionality offered by the component itself is referred to as the *provided function set*.

To allow this exchange of functionality between components different types of communication (message based) channels are offered:

- *send*: this type of channel can only send messages.
- *receive*: this type of channel can only receive messages.
- *request*: this type of channel can both send and receive messages. The only messages it can receive are replies to messages it has previously sent.
- *reply*: this type of channel can both send and receive messages. The only messages it can send are responses to messages it has previously received.
- *duplex*: this type of channel can both send and receive messages without any restrictions.

ECloud makes a distinction between *provided* and *required* channels. *Provided* channels are used to access the functionality of the component while the *required* channels are used to access the functionality of other components. This is related to the distinction which was made earlier between the *provided function set* which is linked to the *provided* channels, and the *dependency set* which is linked to the *required* channels. An example of how to define the channels in a manifest is given in Appendix A (on page 38).

Apart from channels, a component also needs to be able to declare configuration settings which are used when executing the instance of a component. There is a distinction between configuration resources (for example cpu, cores, memory) and configuration parameters which refer to application data. The configuration parameter types offered by ECloud are: boolean, integer, json, list number, string, vhost. If no type is specified, json is assumed. The developers can also chose to define new types which derive from the ones offered. An example of how to define configuration parameters in a component manifest is given in Appendix A (on page 38)

The final element which is needed in order to define a component is the *runtime*. In order for a component to work correctly, the environment on which the component will be run must have certain characteristics, and this is defined by the *runtime*. ECloud provides predefined *runtimes* but it also is possible to define new ones.

As a result, the manifest which specifies the definition of a component includes the component's identifier, the runtime, the configuration and the channels. An example of a complete component manifest is given in Appendix A (on page 38).

3.4 Service applications

According to the ECloud service model, in order for the components to be part of a service application they must be interconnected and play a role. The component manifest defines which channels ca be used communicate, but it is the service application which connects these channels in order to allow the different roles to be able to exchange information.

This is possible due to *channel connectors* which are used to link the actual roles by defining a relationship between their channels. There are three types: *publish-subscribe* (PS), *load balancer* (LB) and *full connector* (FC).

The PS connector links *send* and *receive* channels; messages sent by the *send* channel are received by all *receive* channels (broadcast manner). The LB connector relates *request* and *reply* channels; when a message is sent by a *request* channel, the LB picks one of the instances of the roles with *reply* channel and after the message is processed the same channel is used to deliver the response. The FC connector pairs list of *duplex* channels; in this case when a message is sent by a *duplex* channel from the list, the destination must be specified in order to determine which *duplex* channel from the list receives the message. An example on how connectors are defined in a service application manifest is given in Appendix A (on page 39).

The server application manifest is comprised of the roles, the connectors, the configuration and its propagation to the roles. The service application specifies the configuration parameters in a similar manner to the component manifest, but it also has to specify how to propagate this configuration to the roles. An example of a complete server application manifest is given in Appendix A (on page 39).

4. Simple MongoDB Service

4.1 Introduction

In the previous chapter, some concepts were left out because it is easier to explain them now. Concepts like deployment of a service application, legacy servers (services not designed specifically to work on the native *runtime* of ECloud).

This service topology is just a first design of a MongoDB service, it is not the recommended one because it does not represent the optimal behavior of a MongoDB sharded cluster. The objective of this first approach was to allow the author to learn how ECloud works, test how a legacy server would behave when deployed as a service application on ECloud and how to improve the mechanisms and components which make this deployment possible.

4.2 Topology

For simplicity reasons the replication factor used will be one, meaning that the MongoDB replica sets will be made up of only one *primary* and no *secondary* members. The config replica sets also use replication factor one for the same reason.

Moreover, in this first design, all of the MongoDB components are represented by one ECloud component which includes all the files and necessary logic to run a sharded cluster on a single node. This is the MongoDB component in Figure 6.

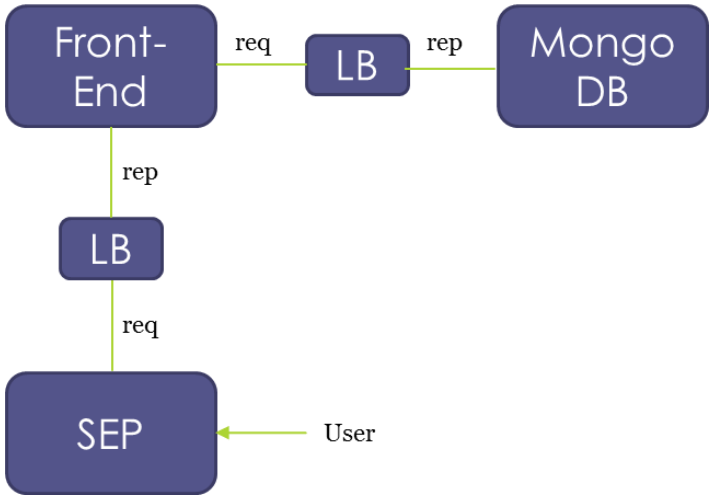

***Figure 6**: Simple MongoDB Service topology*

The MongoDB component offers its functionality by means of a *reply* channel. The component which will access this functionality is the Front-End by using a *request* channel. These two channels are linked by a load balancer connector. The Front-End

component also has a *reply* channel which is used to provide the Service Entry Point (SEP) with the information that the end-user requires. The Front-End and the SEP are connected via a load balancer connector.

The SEP is part of the ECloud built-in services and it allows incoming http connections from the internet to services deployed on ECloud. So in order for the user to be able to use the Front-End, a simple RESTful API can be implemented on the Front-End component.

The Front-End represents the component which implements the MongoDB driver. This driver is used by the user application to interact with MongoDB.

4.3 Legacy servers

All communication between ECloud components is done by using the channels, but the communication between the MongoDB driver and the database itself is done using the Wire Protocol (TCP/IP socket-based, request-response style protocol). For these cases when legacy components expect to communicate directly instead of using the channels, ECloud offers the *proxy-tcp* module.

In order for this to work, both the Front-End and the MongoDB component must use proxied channels and their source code must use the proxy object accordingly. Assume that the MongoDB component wants to have the query router process listening on port 27000. When this component gets instantiated, the proxy will open the port 27000 on a local IP address. The component is aware of this because once this port gets open the proxy fires an *on ready* event which returns the local IP address. Since the port number is known, the legacy server now has the port and IP address on which the query router should listen for requests.

On the side of the Front-End component, a similar situation occurs. This component, once instantiated, awaits for the proxy to notify it with the legacy servers IP and port. Once the legacy server component is listening for new connections, the proxy fires an *on ready* event returning the legacy servers IP and port. This information can then be used to setup the MongoDB driver and connect to the sharded cluster.

5. Complex MongoDB Service

5.1 Topology

The topology of this service relates more to the topology of an actual MongoDB sharded cluster. The *query router*, shard and *config server* logic are implemented in the mongo-router component, mongo-shard component and mongo-config respectively. This introduces more complexity because when the service is deployed, the order in which the role instances are created is random, but the order in which the sharded cluster components have to be added and configured is not. But it also has benefits, more shards can be added to the sharded cluster by instantiating more roles of mongo-shard component thus making scalability possible.

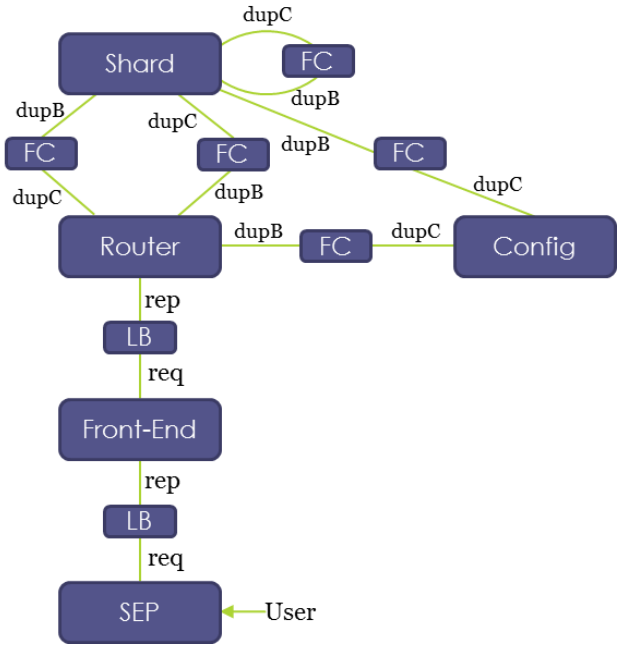

***Figure 7**: Complex MongoDB Service topology*

The topology of the Complex MongoDB Service is represented in the Figure 7. All channels in this figure with the name *dupB* are *require duplex* channels and all the channels with the name *dupC* are *provided duplex* channels. Since MongoDB needs to have an overall view of the sharded cluster, the *duplex* channels paired with the full connectors are used.

5.2 Front-End

The Front-end component contains a RESTful API and a basic MongoDB driver written in coffeescript (based on the NodeJS MongoDB driver [10]) which implements primitive operations such as create, find and delete records inside the database.

This component is very similar to the Front-End in the Simple MongoDB Service. The only difference is that it now communicates with the mongo-router component instead of communicating directly to the sharded cluster.

In order to provide the http connections to the service, the SEP component is used again. The service that creates SEP instances is a built-in service called *HTTP entrypoint*. Even though the SEP acts mainly as a load balancer which redirects incoming http petitions to the legacy server (Front-end component), it must adapt the petitions to send them over the *request* channel. This does not add extra complexity because the protocol used by the SEP (message/http) can be handled by the *http-message* module, which behaves in a similar manner to the *http-server* in nodejs.

The source code for the Front-End component can be found in Appendix B (on page 40).

5.3 Config server

Since parts of the configuration and initialization steps of the three components are similar, with the mongo-config source code being the *base* on top of which the logic for the mongo-router and mongo-shard source code is built, the common parts are explained only in this subchapter.

The mongo-config component is the key component in the sharded cluster. It must be the first component to be up and running, without it the *query router* cannot be started, and without the *query router* shards cannot be added to the cluster. When the service is deployed, the sequence in which the components are instantiated cannot be controlled, so the mongo-router and mongo-shard sometimes will have to be stalled.

The runtime used for all three components of the sharded cluster is the native *runtime* offered by Kumori PaaS. One thing that this *runtime* is missing are the MongoDB binaries, the files needed to execute the actual *mongod* and *mongos* processes. These files will be included in each component alongside the scripts needed to initialize and setup each MongoDB process. Each component in the sharded cluster is responsible for making sure the binaries and the scripts have *execute* permission before anything else.

Both the *mongod* and *mongos* process can be started by either using command-line options or by using at startup time a YAML-based [11] configuration file. This file contains information regarding the storage, log, replication, sharding and networking settings. Most of these settings are predefined, excluding the IP and port. The port is

defined as a configuration parameter in the manifest and the IP is provided by the proxy.

In the case of the *config server*, two *provided duplex* channels are defined, one which allows the *query router* to communicate with it, and one for the shard. The connector used in both cases is the full connector.

The initialization process of the mongo-config component is the following:

- The mongo-config component is instantiated
- The configuration parameters and channels are computed; the configuration parameters provide the port, and the channels are passed to the proxy object.
- The proxy notifies the mongo-config component, by firing an *on ready* event, of the IP which the *mongod* process can bind to.
- The binary files and the startup/configuration scripts are granted the necessary permissions.
- The configuration file is prepared, by adding the missing IP and port needed by the *mongod* to bind successfully.
- The startup script is executed:
    - the necessary directory paths are created for the database and the logs
    - the *mongod* process is executed
    - the config server replica set is initialized

Parts of the source and the startup script are given in Appendix B (on pages 41, 42 and 43).

5.4 Shards

The mongo-shard component is instantiated and configured almost in the same manner as the mongo-config, they are both a *mongod* process, both are a replica set; but the mongo-shard represents the shards which will store the actual data of the database, whereas the mongo-config stores only the metadata since it plays the role of the *config server*. The differences are:

- In the configuration file, the mongo-config has a predefined replica set name and the *configsvr* cluster role; the mongo-shard has the *shardsvr* cluster role assigned and the replica set name is computed by using the mongo-shard component ID.
- The shards must be able to communicate with other shards, this means that the mongo-shard component needs to implement both a *required* and a *provided duplex* channel to allow communication between role instances of this component.

- Apart from the shards communicating with each other, the shard must be able to communicate metadata changes to the *config server*, so a *required duplex* channel is needed.
- Another *required duplex* channel is setup so that the shard can initiate communication with the *query router*.

This topology setup allows the service to operate with more than just one shard, thus providing the scalability which MongoDB is known for.

The mongo-shard itself does not create or add the shards. Even if this component eventually becomes a shard, the configuration done by itself only allows it to reach a replica set state. After all the startup and setup process is complete this node represents a replica set made up of one primary.

The logic necessary for the shard management is inside the mongo-router component.

5.4 Query router

The mongo-router component main concern is to first establish a connection with the mongo-config component. To achieve this, it requires: IP and port on which the *mongos* can bind and the IP and port of the *config server*.

In the case of the *query router* there are two *required duplex* channels and one *provided duplex* channel. The *required* channels are used to connect from *the query router* to the *config-server* and to the shards; the *provided* channel is to allow connections from the shards to the *query router*.

As opposed to the *config server* and shard, there is no replica set running on the *query router*. The *query router* is a *mongos* process and its configuration file needs the networking settings (IP and port to bind to), log settings, the IP and port of the config server. It has no database files stored on it and it needs no extra configuration after the *mongos* process is executed (the *config server* and the shard need to configure their replica set after starting up).

First of all, as soon as the mongo-router component is instantiated, it computes the configuration parameters and it creates the proxy object. All three components share this initial setup, where they take the necessary steps in order to obtain the IP and port necessary to start their respective process, be it *mongod* or *mongos*. But the mongo-router needs more than this in order to startup, it also needs to know the IP and port of the *config server*.

Whenever a new mongo-shard instance is created, the proxy fires an *on change* event which provides the mongo-router component with a list containing the IP and port of all the mongo-shard instances currently alive. If a new instance is born or if an old one dies, the list is updated and sent again. The same applies to the mongo-config instances. This implies that the mongo-router must always check the list and decide if

the instances on this list are shards which need to be removed or added (normally there should be only one mongo-config instance).

This events are asynchronous, so even if the list of mongo-shards available is updated and ready to be processed, no steps can be taken until the mongo-router connects to the mongo-config first.

When the proxy fires an *on change* event the list of instances provided can be related to the *duplex* channel belonging to the mongo-config or the mongo-shard. The mongo-router waits until it receives an *on change* event belonging to the mongo-config, it updates its internal list of available *config servers* and it tries to connect to it. Even though the proxy list has a *config server* this does not mean that the given *config server* is ready to accept connections. This event can be triggered before the instance has finished configuring and starting up. So before attempting to connect to a *config server* the mongo-router uses the *mongo* shell to check when the mongo-config instance replica set has a working *primary*. At first it might get errors regarding that the connection failed because the startup process has not even started on mongo-config instance; secondly the mongo-config might be initializing the replica set and during a period of time it will be in a *secondary* state so all connections attempts from the mongo-router will fail with an error stating that no *primary* was detected for the replica set.

Once the *query router* establishes a connection with the *config server*, new shards can be added to the server. In order to keep track of the shards, an internal list is used. Each time an *on change* event is triggered, it has to be for the *duplex* channel related to the mongo-shard, and the connection details provided in this list must not belong to shards which are available in the internal list and are already added to the sharded cluster. If this criteria is met, the mongo-router tries to add the new shard, or shards.

The same problem as in the mongo-config case arises. The proxy might inform of new mongo-shard instances long before these instances are ready to actually accept connections. So before being able to add a mongo-shard instance as an actual shard of the sharded cluster, that instance should be up and running with a designed primary for its replica set.

Another thing to keep in mind is that when adding new shards to a sharded cluster, the information needed is the replica set name along with the IP and port of the *primary* for this replica set. The mongo-shard component uses its ID to create the replica set name. The details which the proxy includes in its list of currently available instances is the ID, IP and port of the mongo-shards. This way the ID can be used to obtain the replica set name without the mongo-shard and mongo-router having to do extra communication.

The complexity of the whole process resides in the fact that it is asynchronous. Depending on the hardware on which it is being run and delays in the network, defining situations, such as how much time a *query router* should wait before giving up on trying to add a shard which is too slow in completing its initiation process, is difficult. Part of the source code for this component is given in Appendix B (on page 43, 44, 45 and 46).

6. Conclusions and future work

Adapting a noSQL database for autoscaling on a PaaS environment is not trivial and it sometimes requires adding extra functionality on behalf of the PaaS system in order for the database to work. For example, if the Kumori PaaS would not provide the *proxy-tcp* module then any communication between the components of a MongoDB sharded cluster would be impossible, and the service would not even work as a simple replicate set, let alone a sharded cluster.

Regarding autoscaling, as it has been show in chapter 5, it is doable and some parts which normally are done by a database administrator can be automated. On the other hand, in order for a collection inside a database to be split among the shards of a cluster, various steps have to be taken: enable sharding on the database, chose an appropriate shard key, use this shard key to split the collection. This is not an impossible task to be automated and there are some solutions: the service could be set up in such a way that all new created databases have sharding enabled; when the user creates a new collection inside his database he should specify the shard key which should be used. Of course this implies that the logic supported by the *query router* has to be improved, but it can be done.

There are also more complex situations. For example if the user already has a populated database and wants to load it directly in the service without providing a shard key (as explained in chapter 2, choosing the right shard key is very important and it greatly affects scalability). This would require more work on behalf of the platform, an analysis of the data available in the database needs to be done in order to attempt to determine a suitable shard key. This also implies that if all this process has to be automated and done by the platform itself, then a built-in service capable of machine learning techniques is needed. This service could also be used then to define patterns depending on which shards would be added or removed accordingly.

This also raises another issue, how would the scenario of deleting a shard be handled. Deleting a shard is more complicated than adding one because of the draining process: when a shard is removed from a cluster, all the data residing on that shard has to be migrated to the remaining shards in the cluster. This implies that once a mongo-shard instance is given the order to shutdown, the time frame needed by the shard to complete its draining and shut down can vary a lot, depending on factors such as: hardware performance, network latency and data set size.

The logic to remove a shard from a cluster should be implemented in the mongo-shard component. The mongo-shard is aware of all the available mongo-router instances by means of the proxy. Each time a mongo-router instance is created the mongo-shard would be notified via *on change* event issued by the proxy. By using a *mongo* shell to connect to one of the mongo-routers from the list returned by the proxy and issuing the *removeShard* command, the draining process would start immediately and a message stating that the draining has started would be returned. If the same command is issued again, the message returned would specify that the draining is ongoing. Once the draining is completed, if the command is run again, the message

returned indicates that the *removeShard* completed successfully. At this point in time the instance for this mongo-shard could be terminated without losing any data.

As specified in chapter 3, there is the possibility to create *runtimes*. A *runtime* could be created that includes the MongoDB binaries. There are also other optimizations which could be applied to the *runtime* in order for the sharded cluster to operate as optimal as possible in the case of a real life production MongoDB deployment:

- Transparent huge pages: disable and turn off defrag.
- Ulimits: the *ulimit* is used in unix systems to limit resources. A low "soft" ulimit can cause *can't create new thread, closing connection errors* if the number of connections grows too high. For this reason, it is extremely important to set both *ulimit* values (soft and hard) to the recommended values. In order for MongoDB to perform correctly, some of the resources controlled by *ulimit* need certain minimum values. The starting points for large systems are:
    - file handles (fs.file-max): /proc/sys/fs/file-max, 98000
    - kernel pid limit (kernel.pid_max): /proc/sys/kernel/pid_max, 64000
    - maximum threads per process (kernel.threads-max): /proc/sys/kernel/threads-max, 64000

- Ensure that system has swap space configured. Allocating swap space can avoid issues with memory contention and can prevent the *Out of Memory Killer* (OOM Killer) on Linux systems from killing mongod in order to free up memory. Given sufficient memory pressure, data may be stored in swap space.
- TCP keep-alive: In case of socket errors between client-server or between members of a sharded cluster, use a shorter TCP keep alive (order of around 120 seconds) instead of default 7200 seconds (common for Linux systems).
- File systems: XFS is recommended and *atime* for the storage volume containing the database files. The *atime* refers to the file access time.

These are just some of the optimizations and future work that can be done. Some of them are easier to implement and some of them are harder, but overall there is still room for improvement.

7. References

[1] P. Graham. *Web 2.0*, November 2005. [Online]. Available: http://www.paulgraham.com/web20.html. Retrieved: August 27, 2016.

[2] C. Garling. *Amazon Goes Back to the Future With 'NoSQL' Database*, January 2012. [Online]. Available: http://www.wired.com/2012/01/amazon-dynamodb/. Retrieved: August 27, 2016.

[3] MongoDB, *MongoDB Manual*. [Online]. Available: https://docs.mongodb.com/manual/. Retrieved: August 27, 2016.

[4] DB-Engines, *DB-Engines Ranking*, August 2016. [Online]. Available: http://db-engines.com/en/ranking. Retrieved: August 27, 2016.

[5] MongoDB White Paper, *MongoDB Architecture Guide MongoDb 3.2*, June 2016. [Online]. Available: https://webassets.mongodb.com/_com_assets/collateral/MongoDB_Architecture_Guide.pdf. Retrieved: August 27, 2016.

[6] P. Anderson, *Web 2.0 and Beyond: Principles and Technologies*, p. 36, CRC Press, Boca Raton, 2012

[7] E. S. Raymond, *The Jargon File*. [Online]. Available: http://www.catb.org/jargon/html/D/daemon.html. Retrieved: August 27, 2016.

[8] P. J. Sadalage; M. Flower, *NoSQL Distilled, A brief guide to the emerging world of polyglot persistence,* pp. 37-38, Addison-Wesley, Boston, 2012.

[9] MongoDB White Paper, *Delivering MongoDB-as-a-Service: Top 10 Considerations,* August 2016. [Online]. Available: https://s3.amazonaws.com/info-mongodb-com/MongoDB_as_a_Service.pdf. Retrieved: August 27, 2016.

[10] MongoDB, *MongoDB Node.JS Driver*. [Online]. Available: http://mongodb.github.io/node-mongodb-native/. Retrieved: August 27, 2016.

[11] C. C. Evans, *YAML Ain't Markup Language*, [Online]. Available:

http://www.yaml.org/. Retrieved: August 27, 2016.

8. Appendix

8.1 Appendix A

ECloud component manifest:

```
{
  "spec":"slap://slapdomain/manifests/component/0_0_1",
  "name":"slap://afraseniuc/components/feproxylb/0_0_1",
  "runtime":"slap://slapdomain/runtimes/managed/nodejs/0_0_1",
  "code":"",
  "configuration":{
    "proxyTcp": "string"
  },
  "provided":{
    "sepdest":{
      "channel_type":"slap://slapdomain/endpoints/reply",
      "protocol":"TBD"
    }
  },
  "required":{
    "req1":{
      "channel_type":"slap://slapdomain/endpoints/request",
      "protocol":"TBD"
    }
  },
  "external":"TBD",
  "profile":"TBD"
}
```

In the component manifest, the *sepdest* is a provided channel. Since its purpose is to provide functionality it makes sense that the type of channel used is a *reply* channel. The other channel defined in the example is *req1*, a required channel. This is the channel which will allow the component to request functionality from another component, thus the type of this channel is *request*.

The *configuration* parameter represents a string type configuration parameter.

ECloud service application manifest:

```
{ "spec":"slap://slapdomain/manifests/service/0_0_1",
  "name":"slap://afraseniuc/services/mongodbsimpleproxylbservice/0_0_4",
  "components":{
    "sep":"slap://slapdomain/components/httpsep/0_0_1",
    "fe":"slap://afraseniuc/components/feproxylb/0_0_1",
    "mongodb":"slap://afraseniuc/components/mongodbsimpleproxylb/0_0_4"},
  "connectors":[{
      "type":"slap://slapdomain/connectors/lb",
      "dependents":[{
          "component":"sep",
          "channel":"sepsource"}],
      "providers":[{
          "component":"fe",
          "channel":"sepdest"}]},{
      "type":"slap://slapdomain/connectors/lb",
      "dependents":[{
          "component":"fe",
          "channel":"req1"}],
      "providers":[{
          "component":"mongodb",
          "channel":"rep1"}]}],
  "configuration":{
    "proxyTcpFe": "string",
    "proxyTcpMongodb": "string"},
  "defaults":{},
  "configuration_spread":{
    "fe":{
      "proxyTcp":{
        "function":"ident",
        "parameters":[
          "proxyTcpFe"]}},
    "mongodb":{
      "proxyTcp":{
        "function":"ident",
        "parameters":[
          "proxyTcpMongodb"]}}}}
```

The above manifest displays how a LB connector links two components, *sep* and *fe*. In this case the *sep* is the component which requires information and *fe* is the component which supplies it. The *dependents* and *providers* classification is related to the components channels and it corresponds to the respective *required* and *provided* channels.

The *components* describes the roles which the components will have inside the service application.

The *configuration* represents the configuration parameters. The *fe* component will receive at execution time the configuration parameters described by *proxyTcpFe*. The *mongodb* component will receive at execution time the configuration parameters described by the *proxyTcpMongodb*.

8.1 Appendix B

```coffee
q = require 'q'
MongoClient = require('mongodb').MongoClient
assert = require 'assert'

class MongoDriver
  constructor: (@ip, @port, @logger) ->
    @logger.info "MongoDriver:constructor #{@ip} #{@port}"
    # Connection URL
    @url = 'mongodb://'+@ip+':'+@port+'/test'

  connect:(counter) ->
    @logger.info "MongoDriver:connect"
    return q.promise (resolve, reject) =>
      connectCounter = (count) =>
        if count > 0
          @logger.info "MongoDriver:connect #{count} attempts left."
          MongoClient.connect @url, (err, db) =>
            if err
              setTimeout () =>
                @logger.info "MongoDriver:connect timeout"
                connectCounter(count-1)
              ,1000
            else
              resolve(db)
        else
          reject new Error 'Max connect attempts reached'
      connectCounter(counter)

  insert: (field, value, db) ->
    @logger.info "MongoDriver:insert"
    return q.promise (resolve, reject) =>
      # Insert a single document and close db connection after
      db.collection('tests').insertOne {
          "#{field}":value
        }, {
          w: 'majority'
          wtimeout: 10000
        }, (err, r) =>
        if err
          @logger.error "MongoDriver:insert:db.insert #{err.stack}"
          reject err
          db.close()
        else
          db.close()
          resolve(r)

  find: (field, value, db) ->
    @logger.info "MongoDriver:find"
    return q.promise (resolve, reject) =>
      # Get first document that match the query and close db connection after
      db.collection('tests').find({"#{field}":"#{value}"}).limit(1).toArray (err, docs) =>
        if err
          @logger.error "MongoDriver:find:db.find #{err.stack}"
          reject err
          db.close()
        else
          db.close()
          resolve(docs[0])

  remove: (field, value, db) ->
    @logger.info "MongoDriver:remove"
    return q.promise (resolve, reject) =>
      # Remove one document
      db.collection('tests').deleteOne {"#{field}":"#{value}"}, (err, r) =>
        if err
          @logger.error "MongoDriver:find:db.remove #{err.stack}"
          reject err
          db.close()
        else
          db.close()
          resolve(r.result)

module.exports = MongoDriver
```

The two screen captures on the previous page represent the mongo-driver.

The initialization of the mongo-config component:

```
23  run: ->
24    @logger.info 'MongoConfigComponent:run'
25    super()
26
27    [legacyCfg, channels] = @_computeServerParametersAndChannels()
28    @proxy = new ProxyTcp @iid, @role, channels
29
30    @proxy.on 'ready', (bindIp) =>
31      @logger.info "MongoConfigComponent:run proxy.ready #{bindIp}"
32      legacyCfg.ip = bindIp
33      legacyCfg.iid = @iid
34
35      config = new MongoConfig(legacyCfg)
36      check = new MongoCheck(legacyCfg)
37
38      config.run()
39      .then () =>
40        check.run()
```

The method used to compute the configuration parameters and channels:

```
69  _computeServerParametersAndChannels: () ->
70    method = 'MongoConfigComponent:_computeServerParametersAndChannels'
71    @logger.info "#{method} #{@iid} parameters=#{JSON.stringify @parameters}"
72
73    proxyTcpConf = JSON.parse @parameters.proxyTcp
74    channels = {}
75    for key, value of proxyTcpConf
76      channel = @dependencies[key]
77      if not channel?
78        channel = @offerings[key]
79        if not channel? then  throw new Error "Channel #{key} not found"
80      channels[key] = value
81      channels[key].channel = channel
82    @logger.info "#{method} #{@iid} channels =
83                    #{JSON.stringify(Object.keys(channels))}"
84
85    legacyCfg = {}
86    legacyCfg.ip = null # filled when 'ready' event
87    legacyCfg.port = channels['dupSC'].port
88    @logger.info "#{method} #{@iid} legacyCfg = #{JSON.stringify legacyCfg}"
89
90    return [legacyCfg, channels]
```

The method used to grant execute permission to the binaries and script:

```
22  _setupFilePermissions: () ->
23    @logger.info 'MongoConfig:_setupFilePermissions'
24    return q.promise (resolve, reject) =>
25      command = "chmod 775 #{__dirname}/*.sh && " +
26                  "chmod 755 #{__dirname}/../legacy/mongodb-server/bin/*"
27      exec command, (err, stdout, stderr) =>
28        if (err)
29          @logger.error "MongoConfig:_setupFilePermissions #{err.message}"
30          reject err
31        else
32          resolve()
```

The method used to setup the configuration file and start up mongo-config:

```
34  _startLegacyMongoConfig: (legacyCfg, counter) ->
35    method = 'MongoConfig:_startLegacyMongoConfig'
36    @logger.info "#{method}"
37    return q.promise (resolve, reject) =>
38      try
39        configFile = "#{__dirname}/mongod_cfg_svr.yml"
40        config = yaml.safeLoad(fs.readFileSync configFile)
41        config.net.port = legacyCfg.port
42        config.net.bindIp = legacyCfg.ip
43        fs.writeFileSync configFile, yaml.safeDump config
44        logPath = @_getLog(config)
45        setUp = @_getStartUp(config, configFile, logPath, 'setup')
46        startUp = @_getStartUp(config, configFile, logPath, 'start')
47
48        startUpCounter = (cfg, count) =>
49          if count > 0
50            exec startUp, (err, stdout, stderr) =>
51              @logger.info "#{method} #{count} attempts left"
52              if stderr.split(":").pop() is ' Permission denied'
53                setTimeout () =>
54                  @logger.info "#{method} timeout"
55                  startUpCounter(cfg, count-1)
56                ,1000
57              else if (err)
58                @logger.error "#{method} #{err.message}"
59                reject err
60              else
61                resolve(setUp)
62          else reject new Error 'Maximum attempts reached, cant execute startup.sh'
63
64        startUpCounter(legacyCfg, counter)
65      catch error
66        reject error
```

The method used to setup the replica set for the mongo-config:

```
69  _setupLegacyMongoConfig: (config, counter) ->
70    method = 'MongoConfig:_setupLegacyMongoConfig'
71    @logger.info "#{method}"
72    return q.promise (resolve, reject) =>
73      try
74        setUpCounter = (cfg, count) =>
75          if count > 0
76            exec config, (err, stdout, stderr) =>
77              @logger.info "#{method} #{count} attempts left"
78              if stderr.slice(0,25) is 'exception: connect failed'
79                setTimeout () =>
80                  @logger.info "#{method} timeout"
81                  setUpCounter(cfg, count-1)
82                ,3000
83              else if err
84                @logger.error "#{method} #{err.message}"
85                reject err
86              else
87                resolve()
88          else reject new Error 'Maximum attempts reached, config server offline'
89
90        setUpCounter(config, counter)
91      catch error
92        reject error
```

This is the script used to start up mongo-config and initialize the replica set:

```
# Script for starting and initializing MongoDB config server/replicaset
# $1 - database path
# $2 - configuration file path
# $3 - IP
# $4 - Port
# $5 - system log path
if [ "$6" -eq 0 ]; then
  echo
  echo "Creating folders."
  mkdir -p $1
  mkdir -p $5
  echo
  echo "Starting mongod process!"
  echo
  cd /slap/component/legacy/mongodb-server/bin && ./mongod --config $2
else
  echo
  echo "Initializing Config Server Replica Set."
  echo
  echo mongo $3:$4
  --eval 'rs.initiate({_id:"ConfigReplSet",configsvr:true,members:[{_id:0,host:"'$3':'$4'"}]})
  cd /slap/component/legacy/mongodb-server/bin && ./mongo $3:$4
  --eval 'rs.initiate({_id:"ConfigReplSet",configsvr:true,members:[{_id:0,host:"'$3':'$4'"}]})
fi
```

This is the method used to initialize the mongo-router:

```
  run: ->
    @logger.info 'MongoRouterComponent:run'
    super()

    [@legacyCfg, channels] = @_computeServerParametersAndChannels()
    @proxy = new ProxyTcp @iid, @role, channels

    @proxy.on 'ready', (bindIp) =>
      @logger.info "MongoRouterComponent:run proxy.ready #{bindIp}"
      @legacyCfg.ip = bindIp
      @legacyCfg.iid = @iid

    @proxy.on 'change', (data) =>
      @_reconfigLegacyServer data
      .then (response) =>
        if response is '1'
          @legacyCfg.configIp = @configs[0].ip
          @legacyCfg.configPort = @configs[0].port

          router = new MongoRouter(@legacyCfg)
          check = new MongoCheck(@legacyCfg)

          router.run()
          .then () =>
            check.run()
            @readyRouter.resolve()
      .fail (err) => @logger.error "MongoRouterComponent:proxy on change error #{err.stack}"
```

This is the method used to decide which steps need to be taken in case the list provided by the proxy contains either shard IPs or config server IPs:

```
_reconfigLegacyServer: (data) ->
  method = 'MongoRouterComponent: _reconfigLegacyServer'
  @logger.info "#{method} data=#{JSON.stringify data}"
  return q.promise (resolve, reject) =>
    try
      addShard = (data) =>
        @readyRouter.promise.then () =>
          @logger.info "#{method} addShard"
          try
            for newShard in data.members
              newS = JSON.stringify newShard
              count = 0
              for oldShard in @shards
                oldS = JSON.stringify oldShard
                if newS is oldS
                  count = count +1
              if count is 0
                @shards.push newShard
                @_addShards newShard
                .then () ->
                  @logger.info "#{method} added shard: #{JSON.stringify newShard}"
                .fail (err) ->
                  @logger.error "#{method} _addShards error: #{err.stack} "
            resolve()
          catch e
            reject e

      if data.channel is 'dupRC'
        @logger.info "#{method} config server detected: #{JSON.stringify data.members[0]}"
        if @configs.length is 0
          @configs.push data.members[0]
          resolve('1')
          "#{method} config server added: #{JSON.stringify configs[0]}"
        resolve()

      if data.channel is 'dupRS'
        @logger.info "#{method} new list of available shards: #{JSON.stringify data.members}"
        addShard data

    catch e
      reject e
```

This is the method used to add shards to sharded cluster:

```
_addShards: (shard) ->
  method = 'MongoRouterComponent:_addShards'
  @logger.info "#{method} shard=#{JSON.stringify shard}"
  return q.promise (resolve, reject) =>
    try
      shardIp = shard.ip
      shardPort = shard.port
      shardRsName = @_getRS shard.iid

      command = "#{__dirname}/../legacy/mongodb-server/bin/mongo \
                 #{shardIp}:#{shardPort} \
                 --eval 'rs.status().myState'"

      addShard = "#{__dirname}/../legacy/mongodb-server/bin/mongo \
                  #{@legacyCfg.ip}:#{@legacyCfg.port} \
                  --eval 'sh.addShard(\"#{shardRsName}/#{shardIp}:#{shardPort}\")'"

      startUpCounter = (cfg, countConnection, countPrimary) =>
        if countConnection and countPrimary > 0
          exec cfg, (err, stdout, stderr) =>
            if stderr.slice(0,25) is 'exception: connect failed'
              setTimeout () =>
                @logger.info "#{method} replica set offline"
                @logger.info "#{method}, #{countConnection} attempts left"
                startUpCounter(cfg, countConnection-1, countPrimary)
              ,3000
```

```
171                 else if err
172                   @logger.error "#{method} #{err.message}"
173                   reject err
174                 else if stderr
175                   @logger.error "#{method} #{stderr}"
176                   reject stderr
177                 else if stdout.slice(-2,-1) is '1'
178                   @logger.info "#{method} #{stdout}"
179                   exec addShard, (err, stdout, stderr) =>
180                     if (err)
181                       @logger.error "#{method} #{err.message}"
182                       reject err
183                     else if stderr
184                       @logger.error "#{method} #{stderr}"
185                       reject stderr
186                     else
187                       @logger.info "#{method} #{stdout}"
188                       resolve()
189                 else
190                   @logger.info "#{method}: No primary detected for the RS"
191                   @logger.info "#{method}, #{countPrimary} attempts left"
192                   setTimeout () =>
193                     startUpCounter(command, countConnection, countPrimary-1)
194                   ,3000
195             else reject new Error 'Maximum attempts reached, replica set offline'
196
197           startUpCounter(command, 20, 20)
198         catch e
199           reject e
```

This the method used to start the mongo-router:

```
31    _startLegacyMongoRouter: (legacyCfg) ->
32      method = 'MongoRouter:_startLegacyMongoRouter'
33      @logger.info "#{method}"
34      return q.promise (resolve, reject) =>
35        try
36          configFile = "#{__dirname}/mongos.yml"
37          config = yaml.safeLoad(fs.readFileSync configFile)
38          config.net.port = legacyCfg.port
39          config.net.bindIp = legacyCfg.ip
40          config.sharding.configDB = 'ConfigReplSet/'+legacyCfg.configIp+':'+legacyCfg.configPo
41          fs.writeFileSync configFile, yaml.safeDump config
42          logPath = @_getLog(config)
43          startUp = @_getStartUp(config, configFile, logPath)
44          config = "#{__dirname}/../legacy/mongodb-server/bin/mongo \
45                    #{legacyCfg.configIp}:#{legacyCfg.configPort} \
46                    --eval 'rs.status().myState'"
47
48          startUpCounter = (cfg, countConnection, countPrimary, countFilePermission) =>
49            if countConnection and countPrimary and countFilePermission > 0
50              exec cfg, (err, stdout, stderr) =>
51                if stderr.slice(0,25) is 'exception: connect failed'
52                  setTimeout () =>
53                    @logger.info "#{method} config server offline"
54                    @logger.info "#{method}, #{countConnection-1} attempts left"
55                    startUpCounter(cfg, countConnection-1, countPrimary, countFilePermission)
56                  ,3000
57                else if stderr.split(":").pop() is ' Permission denied'
58                  setTimeout () =>
59                    @logger.info "#{method} "
60                    startUpCounter(cfg, countConnection, countPrimary, countFilePermission-1)
61                  ,1000
62                else if stderr
63                  logger.error "#{method} #{stderr}"
64                  setTimeout () =>
65                    startUpCounter(cfg, countConnection-1, countPrimary, countFilePermission)
66                  ,1000
```

```
67                else if err
68                  @logger.error "#{method} #{err.message}"
69                  reject err
70                else if stdout.slice(-2,-1) is '1'
71                  @logger.info "#{method} config-server:#{stdout}"
72                  exec startUp, (err, stdout, stderr) =>
73                    if (err)
74                      @logger.error "#{method} #{err.message}"
75                      reject err
76                    else if stderr
77                      @logger.error "#{method} #{stderr}"
78                      reject stderr
79                    else
80                      @logger.info "#{method} #{stdout}"
81                      resolve()
82                else
83                  @logger.info "#{method} No primary detected for config server replicaset"
84                  @logger.info "#{method}, #{countPrimary-1} attempts left"
85                  setTimeout () =>
86                    startUpCounter(cfg, countConnection, countPrimary-1, countFilePermission)
87                  ,1000
88              else reject new Error 'Maximum attempts reached, config server offline'
89
90            startUpCounter(config, 20, 20, 10)
91          catch error
92            reject error
```