

UNIVERSITY PIERRE ET MARIE CURIE

INTERNSHIP FINAL REPORT

MASTER'S DEGREE

Drivers and Tools at DataStax

Author:
Kevin GALLARDO

Supervisors :
University : Sebastien
MONNET
Company : Shannon
DENNIS

August 22, 2015

Chapter 1

Introduction

This inter-report describes the content of this internship made at *DataStax*, UK, London. The internship was supervised by Shannon Dennis (Senior Director Engineer) and Olivier Michallat (Software Engineer) on the *DataStax* side, by Sebastien Monnet (Associate Professor) and Julien Sopena (Associate Professor) on the University side.

1.0.1 Summary

I've been joining the Drivers and Tools Developers Team. In this report I will first make a description of *Apache Cassandra* including some additional research work on software technologies used by *Apache Cassandra*. Then, I will present the first topic I've been working on, the *DataStax Java Driver* and enumerate the tasks I've been assigned to. I will then describe the second topic of the internship, *Cassandra Detection Tool*. The following part will explain the last big topic of this internship, which is the DSE Graph integration in the *DataStax Java Driver*. I will finally conclude by explaining some miscellaneous additional work related to the 3 big topics I've been working on, and conclude.

Chapter 2

Apache Cassandra

Apache Cassandra is a distributed No-SQL database created by *Facebook* providing high availability and scalability, developed to handle heavy workload produced by their large amount of users and data. *Apache Cassandra* has a master/slave-free architecture which allows it to provide linear scalability and helps instant recovery on hardware and software failures.

The purpose in the following is not to make an entire technical description of *Apache Cassandra* since on the one hand, the internship doesn't involve me on the internal development of *Apache Cassandra*. On the other hand, understanding all concepts of *Apache Cassandra* is important to be able to handle correctly the interaction with an *Apache Cassandra* cluster and thus, be as efficient as possible in the development of the *DataStax Java Driver*.

2.1 General Concept

With *Apache Cassandra*, each node on a cluster of machines is responsible of a part of the data. Thus, a set of data is represented by a Partition Key, and this Partition Key indicates on which node it will be stored, since at each node is attributed the responsibility of a range of Partition Keys. The data can (and is supposed to) be replicated among other nodes with a coefficient of replication that involves different policies of consistency and load balancing. All *Apache Cassandra* nodes will then refer to these policies to manage data among all other nodes.

Writing on a replicated data introduce consistency challenges and is handled in *Apache Cassandra* by broadcasting the `write` operations on all replicas and then waiting on a quorum of replicas to confirm that changes have been applied. A default setting on an *Apache Cassandra* node is to set this quorum's size to $N/2$ (N is the number of nodes owning a replica of the data). Reads on replicated data can be configured according to the client's requirements. Each Cassandra node is aware of all the other nodes, and each node is responsible of the replication of the data in his range of Partition Keys.

2.2 Advanced Description

The following section will describe more precisely some of the key features that allows *Apache Cassandra* to achieve its high availability and scalability characteristics. We will also make some comparison with other database systems that have different behaviors regarding these characteristics.

2.2.1 Transactions

Apache Cassandra version 2.0 introduced a system to handle distributed transactions which, for instance allows clients using CQL operations like :

Listing 2.1: CQL Update Example

```
1 UPDATE ExampleTable
2 SET variable = NEW_VAL
3 IF variable = OLD_VAL;
```

This *IF* condition may not be a big constraint when developing a relational database, but with a fully distributed database, this precondition causes several complications :

- How to achieve linearizability and consistency with distributed transactions ? Use Consensus.
- How to keep an acceptable throughput with a Consensus algorithm ? *Paxos*[1].

But, building these types of transactions involved the need to add some extra-phases to the original *Paxos* algorithm. Here we won't recall *Paxos*'s proofs that define the Consensus guaranties when a majority of nodes responds positively in the *Accept* phase. But original *Paxos* algorithm allows to agree on one value that won't be changed, besides that, what we want to do in *Apache Cassandra* is to achieve linearizability on a suite of operations (or we can say 'values'). Nodes would then be sure of the consistency of the data since every node would have executed operations on its replica in the same order.

To do so, a node must find a way to 'restart' the algorithm in order to make a new decision on another value and chain the algorithm executions. First way to do that would be to build a distributed edit log file based on successive Consensuses made with *Paxos* (a brief review of *Google*'s solution to do that later). To chain *Paxos* execution in *Apache Cassandra* the idea has been to add an extra phase to the algorithm called *commit* phase which simply sends *commit* messages to all nodes to notify them the end of the algorithm.

Another point about using *Paxos* algorithm in our context is that our need is to execute a *compare-and-set-like* operation, whereas *Paxos* algorithm allows to only *set* a value without making the comparison.

Making this possible implied adding another extra-phase to the original *Paxos* algorithm, which is, after receiving a majority of responses from the *prepare* phase. The newly decided coordinator will send a request to all nodes to get their local value

of the replica. If the expected value match, the node proceeds to the following of the algorithm (*accept*), if not, the algorithm stops and the value is not updated. With these solutions, *Apache Cassandra* simply implements *compare-and-set* distributed operations on multiple variables sequentially with the guaranties of the *Paxos* algorithm. With the cost of the overhead produced by the 2 extra-phases.

Google's distributed edit log file

Google's approach to build a distributed fault-tolerant database based on *Paxos* was described in their paper *Paxos Made Live*. It actually explains that even if *Paxos* algorithm could be easily described in one page of algorithmic instructions, engineering a production-ready software based on it, is not so simple.

Indeed, to be fault-tolerant an application has to rely on the hypothesis that a machine doesn't have unlimited memory, and that it can suffer of hard disk failures. Also, the implementation has to be adaptable to different hardware infrastructures since real systems are rarely specified precisely.

The approach taken here was to build a *fault-tolerant* log file that will be describe the operations to be applied on a replica of data. All the replicas hosts would then agree with *Paxos*'s Consensus on the content of this log file and each one would keep locally a version of it. Building a file log of sequential operations implies to chain *Paxos* executions, which are called *instances* (the same idea described earlier for *Apache Cassandra*).

In order to increase throughput and performance of the *instances*, two additional concepts came up in the paper :

- When a coordinator sends data in an *Accept* message, it can concern not only one variable, and update multiple set of variables. So when the *Accept* message is acknowledged by a majority of nodes, all the values will be modified at the same time.
- To avoid as much *Propose* messages overhead as possible, the algorithm could state that a coordinator, once it has passed the *Accept* phase with his value on a previous instance, can stay *elected* for certain period of time. And thus, it can chain *Accept* phases for as long as he is coordinator. He is then called *master*. After that, when a node starts a new instance of *Paxos* algorithm, it first tries to grant a lease from the coordinator (*master*) of the last instance and sets it as his coordinator for this new instance.

Master leases doesn't corrupt the *Paxos* algorithm since a *master* is elected it can fail, and after a certain period of time (*heartbeats*), a new instance will be run and will elect a new *master*.

Besides that, *master leases* improve performance with the fact that when a *master* is elected by a majority of replicas, the *compare* of a *compare-and-set* operation on a *master* can be done locally since once it knows that it has been elected, the *master* knows no other node is going to acknowledge *Propose* messages from another coordinator. Each time a *Paxos* instance is ran, replicas *grant* a lease from the *master* of the previous instance. For *Google*, this implementation of *Paxos* was used as backend of *Chubby*, a *fault-tolerant master-slave* distributed locking system which during its computation stores *Chubby* lock files in a distributed database with replication, and that was then, using this implementation of *Paxos*. This way, *Chubby masters* and *Paxos masters* could be correlated to improve throughput of the application.

Since the log file concerns actions to be applied on a certain data structure, the data structure's history has to be saved, so as for the log file, in case of when a replica fails and has to recover. To handle limited memory restrictions, persistent state's history has been achieved by making *snapshots*. With a proper *snapshot* algorithm the data could be saved in a period of time related to the memory capacity and the log file can also be truncated since it only needs to contain the operations executed since the last *snapshot*.

Implementing these concepts and other technical details allowed to build distributed fault tolerant database relying on *Paxos*.

2.2.2 *Apache Cassandra* local persistence

To achieve local persistence of writes on data in the most efficient way, *Apache Cassandra* rely on some principles that helps acquiring better performance and safety. Each write operation involves a new entry in a local commit file. The write in this file is persisted and it is assumed in *Facebook's* paper on *Cassandra* that this commit file is stored on a dedicated hard disk used only for this purpose, to maximize throughput in regard of the sequential constraints of this one.

To optimize response time on a *read* request, *Apache Cassandra* first does a lookup in the in-memory data structure of most recent data, and if not present, do a lookup on persistent storage. Data is stored in many files on a disk, (ordered from newest to oldest) each part of the data represented by a key. To prevent browsing files that do not contain the needed key, each file owns an index which describes all the keys the file is composed of. This index is called *bloom filter*.

2.2.3 Gossip membership protocol and possible ATTACKS

With *Apache Cassandra*, since the data is distributed and replicated, each node maintains locally the situation it is aware of each other node in the *ring* (the *ring* represents the nodes organized according the distribution of the tokens).

To maintain a coherent distribution of the tokens, in case of when a node fails, a membership protocol called *Gossip protocol* runs regularly to keep the ring coherent. In short, this protocol consists of : an *initiator* node starting a *gossip* round by sending a *receiver* node his current view of all the nodes. The *receiver* updates his view according to the initiator's view (assuming that there's a additional technical process that can insure that a view is more recent than another). And if the view also needs to be updated on the *initiator*, the *receiver* sends the updates in an *ack* message. The *initiator* then updates his views and acknowledges them to the *receiver*. After explaining that this protocol brings a great benefit to the consistency of the system, we will see that some experiences proved that in some cases it can also become a weakness...

This[3] paper explains the vulnerabilities to such a protocol, stating that, in the *gossip* protocol, there is no concern about the about what could happen if the information in these *gossip* messages are false, corrupted. With this statement, we can imagine 2 types of attacks involving a byzantine node that's purpose is to send wrong information among the network.

A first *attack* would be to make some nodes (even maybe all) believe that a node is *down* while he is not. Sending a message to a *receiver* node, falsely up-to-date, saying that a node is *down* can lead the *receiver* node to update his view with a wrong

information. The second *attack* is the opposite of the first one, a byzantine node waits for a node to fail, and while the node is failed, the byzantine one sends *gossip* messages to every other saying that for it (the last updated view) the node is still up. This kind of *attacks* can have a big impact on the performance of the database when there's big loads of data to handle.

It could in the first case, contradict load balancing policies by lowering the usage of the *faked* node, in the second case, involve a great number of lost requests. The paper reports a percentage of up to 83% of lost requests when disseminating up to 75% byzantine nodes in the network, using the *ALL* consistency level. It also describes a *relatively low* performance impact solution to the excess of trust in the *gossip* protocol. It consists in adding an encryption layer to the protocol, allowing to insure that information provided is not from a byzantine node. A little bit like Byzantine *Paxos*[4], with the cost of encryption.

Chapter 3

Java Driver for Apache Cassandra

THE first and main topic of this internship have been to contribute to the open-source project *DataStax Java Driver*. Thus, I will describe how the *DataStax Java Driver* represents more than a simple software that sends packets to a server, by explaining some of its most important and useful characteristics. I will after expose some notable features I've been helping adding in the driver, then present the features I have been working on by myself.

3.1 Introduction

The *DataStax Java Driver* is a high level library that handle interaction with a *Apache Cassandra* database. The *DataStax Java Driver's* main goal is to successfully establish connection to a node, send requests and receive results. It is closely tied to the *Cassandra Query Language* and communicates with *Apache Cassandra* using a communication protocol named *Native Protocol*.

3.2 Basic concepts

3.2.1 Connection

The first step, to successfully achieve communication with a *Apache Cassandra* database is to be able to successfully connect to it through the network. With the *DataStax Java Driver*, it is possible to connect to any of all the nodes in the *Cassandra* cluster by providing at least one IP address of one node.

The main representation of the connection to a *Apache Cassandra* cluster is exposed through a *Cluster* object. To optimize latency, throughput and scalability (more on that later), the driver will then gather a lot of information from the node it is connected to. The first node connected, becomes what we call the *Control connection*. After connecting to the first node, the driver will execute several additional tasks :

- Connect to all other known up nodes in the cluster.
- Collect cluster metadata.

- Collect schema metadata.

After these tasks successfully executed, a *Session* object is returned to the client. This *Session* object is the object from which the client will be able to send, prepare, and bind CQL statements.

3.2.2 CQL

The CQL query language is a database query language specific to *Apache Cassandra*. Its purpose is to fit the most possible to the existing universal SQL query language, and in the same time provide *Apache Cassandra* specific syntax. CQL is then designed to provide options and tools to handle distributed datasets, with the possibility to manage consistency levels, replication factors, data partitioning, node management, and so on. CQL statements are sent by the driver to a *Apache Cassandra* cluster through the *Native protocol*.

3.2.3 Native protocol

The *Native protocol* is a network applicative protocol built on top of *TCP*. It has been designed to be fully asynchronous, hence, on the driver side it allows to accumulate queries on a single connection in an asynchronous manner. Indeed, each request on a connection gets assigned to a request identifier, named stream id, created in a thread-safe way on the client side. Multiple requests can be sent asynchronously and concurrently through the connection, each response coming back will be assigned to the stream id it has been issued from. The driver also adds a client-side timestamp on each request, to insure order of requests when needed by the user. *Native protocol* also allows security functionalities like server-side authentication and SSL encryption. Most of the work accomplished by the *DataStax Java Driver* is to encode and decode data sent and received through the *Native Protocol*.

3.2.4 Processing responses

The *Native protocol* being fully asynchronous, a client has the possibility the process the results of its queries either synchronously or asynchronously. Internally, everything is handled asynchronously, every request implies the creation of a *Future* object from the Java's *Future* library. On these *Future* objects we can define callbacks to fill the data when a connection notifies that the data has arrived through the connection. Since internal processing in the driver is asynchronous, most part of the code is thread-safe.

3.2.5 Metadata

Aside from sending statements, and processing results, the *DataStax Java Driver* exposes a complete description of the cluster, schemas, tables, indexes, partitions, through the *Metadatas*. This metadata is computed at the *Cluster*'s initialization and is maintained up-to-date all along the *Cluster* object life span. Having this metadata improves the driver knowledge of the current status of the system, and allows the driver to adapt it's behavior according to it. The metadata is also exposed to clients through API specific objects.

3.3 Advanced Features

In this part I will try to explain some more advanced features that contribute in making the *DataStax Java Driver* more powerful, scalable, tunable, and probably justifies the 70.000+ public downloads of the software each month.

3.3.1 Connection Pooling

As stated before, at cluster initialization, the driver establishes connection to the first IP address that has been specified, and this node becomes for the driver, the *Control connection*. In addition to that, the driver will also establish a connection to all the other nodes in the cluster. This is made for multiple purposes. First, in case of a clear failure on a node, which doesn't respond in time (timeout), or returns an internal exception, so the driver still has a way to process the request for the client. Remember that *Apache Cassandra's* architecture is Master-less, which means every node is able to handle every request, the nodes are aware of the data repartition (Token routing) and replication. The second advantage of connecting to all nodes is load balancing. Indeed, to handle heavy loads needed for some applications, the driver is able to distribute the requests to all nodes equally, so that one node connected doesn't get overloaded (more information in the *Policies* section).

But that's not all, to achieve even better throughput the driver establishes pools of connections for each host. This is needed mainly because requests can take a variable amount of time to be executed on the server side, and since everything is asynchronous, the protocol accepts adding pending requests on a same connection (thanks to stream ids). But it has its limits, the *Native protocol* version 2 can only handle a maximum of 128 simultaneous requests on a same connection. This limit can quickly transform into a bottleneck, that prevents achieving high throughput. That is why the driver has pools of multiple connections for each node.

Unfortunately achieving high throughput comes with the price of having to tune and maintain the connection pools. Thanks to the highly tunable nature of the driver, users are able to define rules to dynamically resize connection pools according to the current load. Indeed when the load becomes more consequent, the driver automatically increases the number of connections in the pools. We can detect that the load becomes heavier because the driver maintains a number of pending requests per each connection. Hence, the decision to open a new connection is made when all existing, except one, connections in the pools have the maximum allowed pending requests, and when the number of pending requests on the last connection becomes higher than a defined *threshold*.

The driver provides the option to set the original number of connection per pool, the maximum number of connections to create, and set the *threshold* to use for connection creation, and also set the *threshold* indicating whether to close a unneeded connection.

With fine tuning of these parameters, our experiments prove that the driver is able to issue comfortably a total of 100k requests/node.

In more recent versions of *Apache Cassandra*, the *Native protocol* have been updated and is now able to support 32768 requests on each connection. This greatly helps improving the usage of a connection pool, unfortunately experiments still show a bottleneck in using a limited number of connections in pools, more on that later, in the *Netty* section.

3.3.2 Policies

The Java driver is highly tunable, and some of its most important settings are defined with policies. A policy is a Java object defined by the user, implementing an *Interface* of the driver, and whose methods will be injected used in the driver's code. The user can specify the policies during the cluster's creation.

Load Balancing Policy

The driver provides a set of pre-configured and highly tested load balancing policies. However, a user can inject its own. The load balancing policy is the class defining the *query plan*. The *query plan* is the object which defines what will be the host chosen to send each query, it's a Java *Iterator* which will return a host each time the driver needs to send a request. This setting is important in order to distribute the load on as much nodes as possible and thus, avoid bottlenecks and improve throughput. For example, the most common and meaningful balancing policies to use, for normal use-cases is the *TokenAwarePolicy* with a *DCAwareRoundRobinPolicy* fallback option. It is set constructing the following object a cluster's creation :

Listing 3.1: Creation of the cluster

```
1 Cluster.builder()
2     .withLoadBalancingPolicy(new TokenAwarePolicy(new
        DCAwareRoundRobinPolicy, true));
```

This setting allows to send statements to directly to hosts having a local replica of the data according to the routing key of the statement (that's the goal of the *TokenAwarePolicy*) and if no local replica was found, send the statement in a round-robin fashion, to hosts from the same DataCenter (the role of the *DCAwareRoundRobinPolicy*). Other useful Load balancing implementation are provided from the driver such as *WhiteListPolicy*, or *LatencyAwarePolicy*. But users can still implement their own policy and inject it in the driver. By default the driver uses a *RoundRobinPolicy* based on its knowledge of the hosts located in Cassandra's table *system.peers*.

Retry policy

The retry policy defines what should be the behavior of the driver in case a query sent to a Cassandra node happens to fail, and the node originally contacted for the request sends back to the driver the error corresponding.

Executing a query can fail in two ways :

- The error happens on the driver side. The driver is not able to communicate to the server in the required time, or no host is available anymore, a network failure occurs.
- The coordinator (host contacted for the query) is reachable for the driver, and the coordinator receives the query, but an issue internal to the processing on the server-side occurs. This can be a consistency level issue, a network issue between multiple Cassandra nodes, and so on.

For the first case, the driver provides an API to tune the socket options. With this, users can set timeouts, buffer sizes, etc. The second point is handled differently,

when a server-side issue occurs, the coordinator will be able to send an error message to the driver. Then, the reaction of the driver according to different kind of errors will be defined by a *RetryPolicy*. The errors returned can be of 3 types :

- Read timeout
- Write timeout
- Unavailable host

Each of these errors is likely to happen because the consistency level required for the query could not be met (e.g if 3 replicas were required but only 2 responded within the defined server-side timeout). Then, the driver will call the corresponding method in the *RetryPolicy* object it has been constructed with. If the error is a *ReadTimeoutException*, the driver will call *RetryPolicy.onReadTimeoutException()*.

As usual, the driver provides highly tested and pre-configured Retry policies. An interesting one is the *DowngradingConsistencyRetryPolicy*. Thanks to this policy, we can define that when a *ReadTimeoutException* happens, the driver will re-send the request, to the same host, with a consistency level reduced by 1 and only once, otherwise a Java exception is thrown.

Reconnection policy

A reconnection policy is meant to be used in cases where a connection in a pool gets closed because of a particular error. The reconnection policy mainly helps define the time before the next reconnection attempt gets scheduled. Users can explicitly set this time, or *ConstantReconnectionPolicy* can help define a constant time before reconnection, and *ExponentialReconnectionPolicy* helps define a base delay which will grow exponentially, until a maximum delay is reached, then when the maximum is reached, it will stay constant.

3.4 Object Mapper

Chapter 4

Achieved work

The main purpose of this internship at DataStax is to contribute to the active development of the open-source *DataStax Java Driver* for *Apache Cassandra*. This includes aggregating the client's API with new functionalities to fit *Apache Cassandra* functionalities, internal operation of the driver, improving Query Builder¹ and Object Mapper² functions, resolving reported internal driver's issues and suggestions. I will then, give a brief description of some of the most significant tasks I have been working on since now. I have also been assigned to build a new *DataStax's* tool called *Cassandra Detection Tool* (described later).

4.1 Java-Driver

4.1.1 Refactoring Integration Tests Class

The test architecture for the *DataStax Java Driver* is composed of both unit and integration test. Those test are ran 24/7 for continuous integration by *Jenkins* servers. To simplify building these tests, a big number of integration tests are classes that extend an *all-configured* class which has the responsibility to create the necessary connections and configurations with a test cluster using DataStax's specific tools for automating *Apache Cassandra* cluster creation.

The previous behavior was that inheriting this all-configured implied the need to recreate a new cluster each time. The process was simplified by creating a generic class that initiates a cluster and stays instanciated for all classes inheriting it, adding a thread-safe need for this class. Integration tests suite runs approximately 30% faster.

4.1.2 Manual Paging

Requests on *Apache Cassandra* that generate a large amount of data are transfered in network applicative frames called *pages*. These pages are identified by *Apache Cassandra* and are communicated through the *Native protocol*. The goal of this improvement was to give client the possibility to handle themselves the *paging state*

¹Query Builder is a library developed for the Java Driver allowing clients to build query in an Object Oriented way. Example : `Statement statement = QueryBuilder.select().all().from(keyspace, table);`

²Object Mapper is an Object Relational Mapping tool developed for the Java Driver.

of a request. Thanks to his *paging state* the client start fetching some data from the server, stop processing, and re-send a new request but with the *paging state* information which allows him to continue its processing from where he had stopped. Giving the possibility to the driver to be used in a entirely stateless environment. This task involved studying *Native protocol* and gathering information by manually decoding frames, changing the internal use of the *paging state* in the driver and understanding *Apache Cassandra's* usage of the *paging state*.

4.1.3 Asynchronous host pools creation

Until now, creating a pool of connection to a *Apache Cassandra* node was all made sequentially. Moreover, making an authenticated connection on a *Apache Cassandra* cluster implies a certain overhead due to password verification. Therefore, when the driver has to make sequentially a pool of authenticated connections it could imply too much latency on a cluster with a large number of nodes.

The solution was to make the pool creation entirely asynchronous. Thus, all connection pools are made in parallel, and the driver worker only have to wait for all connections to be established. Since we are in a HIGHLY concurrent part of code, lots of issues were produced when adding this feature and needed a big part of refactoring of code. Temporary tests report a improvement 8x the time for the driver to connect to a 40 nodes cluster.

4.2 Cassandra Detection Tool

Cassandra Detection Tool is a software I have been asked to contribute to that's main function is to provide to customers all useful *Apache Cassandra* information we can gather concerning a given a range of IPs. Customers owning a (*very*) large number of clusters among its network sometimes isn't even sure how much nodes are running. *Cassandra Detection Tool* makes then, a clean and concise description of the *Apache Cassandra* instances. A first prototype of the tool has been made by *Martin Van Ryswyk*, our Executive Vice President in Engineering, and my work have been to build a final working solution of the software.

It turned out that I changed almost all the code conserving the original conception design. Technically the process was to first, use asynchronous Java sockets to try to establish connection on all of the given IP, all tries made in parallel. Thanks to that the software could establish a list of UP nodes running a *Apache Cassandra* instance. Moreover, the app use the *DataStax Java Driver* and the *Java Thrift API*³ to connect to running hosts and gather all the node information possible. Information concerns : token map, cluster's hosts, keyspaces and all types of versions. To be more efficient in case of a *big* list of hosts (we can aim for a class B network) to connect to, the application creates a dynamic sized thread pool and then tries to use drivers to gather information. Also, to avoid the need for a thread in the pool (*worker*) to have to, let's say, send gathered information to a main worker that processes all the results and etc... the information was stored in *static* (shared) data structures so it was highly concurrent because a certain number of nodes can be located on the

³Apache Thrift is a network applicative protocol originally used for Cassandra but got replaced by the Native protocol because of performance issues and also because the Java API for Thrift was really hard to use and not well designed. It is still available (but not updated) and Deprecated but I use it here for compatibility with clusters potentially using an old version of Cassandra.

same cluster, and modifying the list of nodes in this *Cluster* object has to be doable concurrently. As a result, the code is thread-safe.

It was great to have the opportunity to build this software on my own, and also I'll have the chance to make a video presentation to clients and sales team people on how to use the software (even though I tried to make a GUI as clean as possible).

Chapter 5

What's next

For the following of the internship I will continue to contribute to the *Java driver* bringing new functionalities and trying to understand more and more of its code in order to be able to handle more easily the development process.

Bibliography

- [1] Leslie Lamport. *Paxos Made Simple*. ACM SIGACT News (Distributed Computing Column), 2001.
- [2] Tushar Chandra, Robert Griesemer, Joshua Redstone. *Paxos Made Live - An Engineering Perspective*. June 20, 2007.
- [3] Leonardo Aniello, Silvia Bonomi, Marta Breno, Roberto Baldoni. *Assessing Data Availability of Cassandra in the Presence of non-accurate Membership*. University of Rome, Italy, 2013.
- [4] Jean-Philippe Martin, Lorenzo Alvisi. *Fast Byzantine Paxos*.