CassandraFS: A Highly Available/Scalable File System over Cassandra[1]

Ftylitakis Nikolaos
ftylitak@csd.uoc.gr

ABSTRACT

The Cassandra is highly scalable second-generation distributed database, bringing together Dynamo's fully distributed design and Bigtable's ColumnFamily-based data model. Our goal is to use Cassandra as a File System creating an API of Cassandra's functionalities. The aspects of consistency and availability will be examined in conjunction with the File System needs.

1. INTRODUCTION

Our times are characterized by the evolution of networks, the spread of processing power and generally a trend to move from the individuality to globalization. Our era is named by many "The Era of Information" so that arouses a challenge. The challenge is to make reliable systems capable of keeping alive that Information in a consistent and fault tolerant way.

Cassandra[1] is a highly available, scalable database-like system initially created by Facebook to implement the Inbox Search operation. It can be though as the merging of Amazon's Dynamo[2] and Google's BigTable[3]. This system will be used as the base to create a highly available, highly scalable File System

Section 2 analyzes the Cassandra's structure, data model and topology. Section 3 shows how the File System was build on top of a Database. Section 4 deals with Consistency issues that aroused. Section 5 is the evaluation of the project followed by Section 6 with the future work and the conclusion in Section 7.

2. Cassandra

Dynamo offered the concept of decentralization. The topology is considered to be a ring of equal nodes without single point of failure. Every node is accessible by the clients and every node can execute any operation making the system highly available. Node failure is considered the common case and not an exception since the increment of scalability results higher rates of failures. That's why the cost of rejoining of a node is intentionally kept low. When a node joins the system, is assigned with a unique hash value and according to it is placed to the ring. Goal is to place the nodes as symmetrical as possible to achieve load balancing. That value is furtherly used to bind a range of keys (hash values) that the node will be responsible to handle. The range starts from the key of the node that is closer and always lesser than the key of the current node. Simply stated, a node is responsible for the keys previous to it but larger than its previous node. Since everything is placed in a ring, values larger than the total key range are rounded.

Every data that is ready to be inserted to Cassandra is assigned with a hash value the same way a node takes place to the ring. That value is the data's ticket for the Cassandra since it states exactly in which node it is going to be placed. Apart from that node, the cluster can define a replication factor for its data. When data are inserted with a replication factor of N, apart from the responsible node, N-1 other nodes will be selected to hold replicas of that data. The other nodes are selected either simply the following N-1 from the current, or for purposes of load balancing nodes less loaded than others. Higher guaranties of data safety are provided that way but also higher throughput since upon data reading from a randomly selected node, the data are retrieved from the closest to it node in the ring.

This arouses issues of consistency. Cassandra is considered to be eventual consistent with tunable consistency level. For the purposes of this issue two mechanisms are supported, Hinted Handoff (like Dynamo) and Read Repair.

Now looking Cassandra from its data model point of view, it is described as a columnar data model. A table in Cassandra is a distributed multidimensional map indexed by a key. The value is an object which is highly structured. The row key in a table is a string with no size restrictions, although typically 16 to 36 bytes long. Every operation under a single row key is atomic per replica no matter how many columns are being read or written into. Columns are grouped together into sets called column families very much similar to what happens in the BigTable system. Cassandra exposes two kinds of column families, Simple and Super column families. Super column families can be visualized as a column family within a column family. The top dimension in Cassandra is called Keyspace. By convention, for each application a new Keyspace is used.

Data access is similar to a 4 or 5 dimensional hash table. The use is with the following order:

1. Keyspace: Intuitively specifies what kind of data you want to access considering that different applications use different Keyspace.
2. Column Family: It is roughly analogous to a Table in a relational model. Because every Column Family is stored in a separate file, Columns that are going to be accessed together should be placed to the same Column Family.

Course project for Modern Topics in Scalable Storage Systems, University of Crete. Computer Science Department Winter 2010

3. Key: The string that specifies a row. Under each key, a separate number of columns can be assigned independently from other keys.

The following separation between Simple Columns and Super Columns is defined at the configuration file of each Cassandra node and can be changed only after restating the instance. Both of them have the ability to sort their data by Date, Alphanumeric comparison, Byte comparison or as Integers.

4. Super Column (optional): A container for an ordered collection of Columns.
5. Column: A structure containing a name, a value and a timestamp. It's the basic level of storage and it's dependent to the key.

The basic API provided is the following:

- get(keyspace, key, column_path, consistency_level)
- insert(keyspace, key, column_path, value, timestamp, consistency_level)
- remove(keyspace, key, column_path, timestamp, consistency_level)

Rest of the functions are batch versions of the above.

3. FILE SYSTEM ARCHITECTURE

Cassandra, as we saw, has great advantages on been a scalable consistent information storage. Creating anything above it immediately inherits all those great features. That's a reason why Cassandra was chosen to build our File System over it. And it was an interesting challenge. How to build a File System over an API of a database-like system. At this point we have to pay tribute to the work that people from project **cassandra-fs[4]** have done. From that project we have got the initial idea and implementation. Using that information we will start to analyze what where the challenges and how it was achieved.

3.1 Challenges

Main interest was where to place the information of the File System. First of all, a new Keyspace has to be selected for our goal and the name was"FS" (hard to imagine). Next step was to decide which Column Families would be used. As we mentioned previously, a Column Family is kept in a single file at the responsible nodes. So data that will to be accessed together is preferred to be placed to the same Column Family. As of that, it was chosen to use the following 2 Column Families:

- *File*: It will contain all the information required to fully describe a file. That would be its *Meta-data* and its *Content.*
- *Folder*: For every folder created, it will contain all the Meta-data of the files that the folder contains

3.2 Storing a File

A file to be fully described has to contain its name, meta-data and its content. All of them are simple values so the *File* Column Family was selected to be Simple Column Family. Every row will represent a file which will divide its information into Columns. Each row is selected by a key so the key needed to be something unique for every file. So as of now, the string key of a file will be the file's absolute path in the file system followed by its name (e.g. "/home/usr/file.txt"). This is unique for every file. Any operation referring to that key is done to that specific file and only.

Since the row is ready we can proceed to the description of the Columns that where used:

- *Content*: here is the heart of the file's data. It is a Byte array whose maximum length if 5MB (configurable). It holds the actual information of the file.
- *Length*: the size of the file in Bytes. It can be bigger than the length of the *Content* (in case we have a file bigger than 5MB but this will be explained in a moment)
- *Last Modified*: The timestamp showing when that file was last modified
- *Owner*: the name of the user that created it. This could be used for permissions but in this version it is not implemented yet.
- *Group*: the name of the group that the owner belongs to. Also will be used for permissions.
- *Attribute Type*: Contains the string "File".

The problem here is that the content can hold as much as 5MB. What happens if a client wants to store a file larger than that size? In that case the same columns are kept, the only difference is that we will use more rows. The file will be splitted into chucks of 5MB and they are going to be placed in sequential rows.

First of all, the key of the first row remains the same as in the case of using one row. So it is kept "/home/usr/file.txt". The other rows use that string as the base for the key in combination with a serial number to identify the sequence of the chucks. This results a string key of the type "/home/usr/file.txt_$ID". The reconstruction of a file upon reading is simply done by merging all the chunks using the above sequencing,

We have explained what keys are used and what is stored in the Column *Content* but what happens with the Meta-data of the file? All rows of a file regardless their number, hold the same Meta-data. That is the overall information of the file. So the Column *Length* doesn't simply holds the length of the *Content* in the same row but the overall length of the file when successfully merged. *Last Modified* Column likewise holds the latest modification time of the chunk that was edited last. This convention is helpful for consistency checking. Having only a chunk the client can access the overall info of the file to be able to compute the overall number of chunks.

The approach of placing the chunks in different rows than in different columns has its pros and cons.

Cassandra has a process of freeing up space called Compaction. It is in charge to merge large accumulated files computing and comparing Merkle Trees of the rows of a Column Family. To create a Merkle Tree of a row, the row has to be loaded to the main memory (RAM) of the system initiated the Compaction. If the row size gets too big there might be memory problem. So

using small chunks in each row makes the Compaction easier and faster.

On the other hand there is the problem of scattering the chunks in different nodes. We already know that data are assigned to a node according to its key. Now let's imagine that we are storing a multi-row file. Every row has a different key so it is highly possible that the chunks will be placed in different nodes. This can increase the overhead of getting the complete file meaning lower read rates.

An idea here would be to create something in the middle. May be we could store a number of chunks in the same row in different Columns with a restriction in the number. So we can keep the chunk size of 5MB and spread at most 100 chunks in a row resulting about 512MB per row. This is not an outrageous number for a single node to load in its main memory at a time, considering the amount of RAM nowadays computers have. The naming of the Columns would be analogous to the naming of the different rows. That would be "Content", "Content_$1", , "Content_$N". Of course if a file is larger than 512MB then extra rows can be used in the same concept as the current implementation.

3.3 Storing a Folder

A folder is responsible of knowing what it contains. It is not obliged to be able to provide us also with the contents of the files it has. During a lookup of a folder's content we want an efficient way of getting that information. That is why we choose a *Folder* to hold a replica of the Meta-data of the files it contains. As we saw, files are represented by the Meta-data Columns and the Column *Content*. To get the Meta-data of a file, we have to keep a different table for each one containing the 5 Meta-data Columns we defined above for the Files. This leads us to characterize the *Folder* Column Family as a Super Column Family. Each row is a folder and the row's key is the absolute path of the folder. The value that is stored to that row is a table. Each row in the subtable is a file in the folder and the columns are its information.

The issue here is the duplicate existence of all files Meta-data. Although this gives us lower lookup times it increases the cost of making the duplicates consistent to changes after many read-write operations.

3.4 File System API

The API provided for using the File System is simple enough.

- cd <folder>
- mkdir <path>
- rm <file | folder>...
- ls <path>
- newfile <file> <content>
- cat <file>...
- copyToLocal <source> <dest>
- copyFromLocal <source> <dest>
- copyFromHDFS <source> <dest>
- copyToHDFS <source> <dest>

The last two functions shows the ability to communicate directly with Hadoop.

4. CONSISTENCY

In this section we will discuss the possible impacts of Cassandra's Consistency operations to our data model of the File System. This issue was brought up because of the operation called Read Repair. It is initiated from every read of data in order to merge different versions of the columns of a row. We will discuss about this furtherly afterwards but let's first explain the Consistency concept of Cassandra.

Consistency is one of the ACID properties (Atomic, Consistent, Isolated, Durable). Its policy describes the legal states our system can be and ensures that after every transaction our system will be to one of those states. Cassandra can be considered eventual or fully consistent depending to the Consistency Level that we define during our transactions.

As we said, every data inserted to Cassandra is replicated to N nodes. The value of N is chosen by each Cassandra instance at start up. We will see that replication factor and consistency are dependant each other.

Table 1 shows the connection between Consistency Level and the replication factor. For write operations, any of them can be used. For reads only the last 3 are possible to be used. Hence that higher Consistency level results lower performance.

Table 1: Consistency Levels (read rows doesn't exists at read operations)

Level	Description
ZERO	Cross fingers
ANY	1st Response (including HH)
ONE	1st Response
QUORUM	N/2 + 1 replicas
ALL	All replicas

Let's see Table 1 in more detail. Suppose a client is ready to execute a write operation to the system and wonders what Consistency Level (CL) to use.

Selecting ZERO CL means that as soon as the write message is send, we continue our normal execution without waiting for acknowledgments from the nodes responsible for the replication of that data. This option of course is the fastest but is the most unsafe since we don't know if the data were even written to one node. This means that upon reading we can't be sure if we will find the data that we tried to write.

ANY CL will make the client wait for exactly one acknowledgement but it is different from the ONE CL. The difference lays to the fact that the one respond that will be received can be from a node having a Hinted Handoff. In a few words a Hinted Handoff happens in case a responsible for a replica node is down (network partition or any other reason), another alive node buffers the messages that were to be received

by the “dead” node. As soon the “dead” node comes back to life the other node passes the messages and the when they are received the resurrected node will execute them. So if a node that has Hinted Handoff replies, the data are still nowhere.

ONE CL is exactly the same as ANY CL but without the Hinted Handoff. Acknowledgements only from nodes responsible for receiving the specific data are accepted. By getting one of those acknowledgements we are ensured that at least one node has our data. From that time and on, the data are propagated in the background, to the rest of the responsible nodes. So at some time if every node is available all nodes will have the data inserted. This is called Eventual Consistency.

QUORUM CL can be parallelized with the elections. If most of the people agree then it is done. Back to Cassandra world, the client waits for N/2 + 1 replicas to respond. It means that if more than half of the nodes responsible for the replica respond success then the operation is successful. A very nice option indeed and in most cases it is more than enough for the client to continue.

Finally full consistency can be achieved using ALL CL. Here all the replica nodes have to respond for the operation to be successful. In any other case an error message is returned to the client.

Taking out the write operations, what happens at reading? ONE, QUORUM and ALL CL affects read operations exactly the same. ZERO and ANY CL can’t exist since the data are not even written yet.

Since we described the Consistency Levels, which can be seen as the expectations that a client may specify we will move forward to see the Consistency operations executed from the side of Cassandra.

4.1 Hinted Handoff

This is a mechanism to ensure availability, fault-tolerance and graceful degradation. If a write operation occurs, and a node that is intended to receive that write goes down, a note (the "hint") is given ("handed off") to a different live node to indicate that it should replay the write operation to the unavailable node when it comes back online. This does two things: it reduces the amount of time that it takes for a node to get all of the data it missed once it comes back online; it also improves write performance in lower consistency levels. That is, a hinted handoff does not count as a sufficient acknowledgement for a write operation if the consistency level is set to ONE, QUORUM, or ALL. A hint does count as a write for Consistency Level ANY, however. Another way of putting this is that hinted writes are not readable in and of themselves.

The node that received the hint will know very quickly when the unavailable node comes back online again, because of Gossip. This mechanism has no sideffects to our data model of the File System because it simply replays the missed operations and results a consistent state. If, for some reason, the hinted handoff doesn't work, the system can still perform a read repair.

4.2 Read Repair

This mechanism is mostly initiated by a read request to some data. At every read the node that got the read request initiates a small check to ensure that all the replicas have the same version of the data. First, queries all other replicas for the digest of the data version they have. A digest is a value generated by hashing the selected data along with its timestamp. By definition, the digest uses a much smaller amount of memory than the original data would, minimizing the amount of data sent over the network for the checking. The digests are compared. If any of them mismatches, the node pulls all data from the replicas and merges them.

The merging takes place to one row at a time. Having more than one version of a row means either that a row has data on different Columns or that outdated data exists to some of the Columns of at least one replica. In the first case, all different Columns found, are placed to the final version of the row. In the second case, the timestamp is checked and the most recent data are kept. Finally, since the most recent version of the row is created, it is written to the replicas whose digest was mismatched.

Although this operation would be acceptable in other cases, in our case it can corrupt our data in extreme cases. Our file system saves chunks of data to rows. A single row is a well defined file system structure containing the files Meta-data and its content. If any of those Columns get changed separately then the row is corrupted and subsequently the file where the chunk belongs. This results loss of data, a definitely not acceptable scenario to our file system.

The Read Repair needed to be changed to ensure our desired consistency. To achieve that, the implementation of Read Repair was edited exactly at the place where the merging takes place.

So we are back at the moment where a node has identified a mismatch, pulled all the replicas and is ready to merge them. We start comparing the replicas data. The replica with the higher number of occurrences is selected as the most valid and is propagated to any node has a different replica.

That way we ensure consistency at the level of each row. There is no merging of Columns. The final replica is created by only one file, not a concatenation of many more.

Our implementation could be improved by using the timestamps of the Columns. Instead of finding the replica with the highest number of occurrences it would be better to select the replica with the latest timestamp. That was our original goal but we unable at this time to retrieve the timestamps out from the columns.

It has to be stated that Read Repair is initiated regardless the Consistency Level of a read operation. If CL is set to ALL then the client would have to wait until the Read Repair finishes and all replicas has the same value. At any other CL, the Read Repair is initiated and executed in parallel with the read operation. The operation might collect the required acknowledgements but Read Repair will continue its execution until eventually the consistency is achieved.

5. Evaluation

To create a real world implementation of the above we used 8 PCs. Four of them were used to host an instance each of Cassandra server. The hardware was 3PCs x Intel Core 2 Duo E6400 2,13GHz, 2GByte Ram running Windows XP 32bit and one Intel Core 2 Quad Q9400 2,66GHz, 6GByte Ram running Windows 7 64bit. These 4 PCs communicate over a 1Gbit

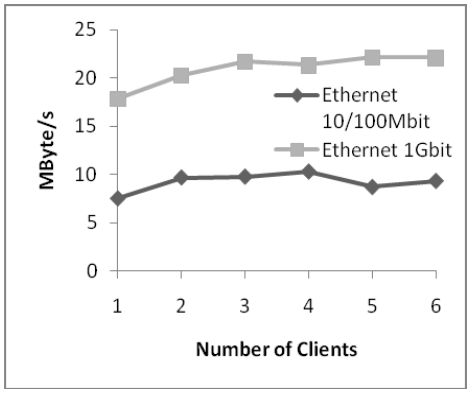

Figure 1: Write throughput under Single Pc Testing, 1 Server PC, 1 Client PC

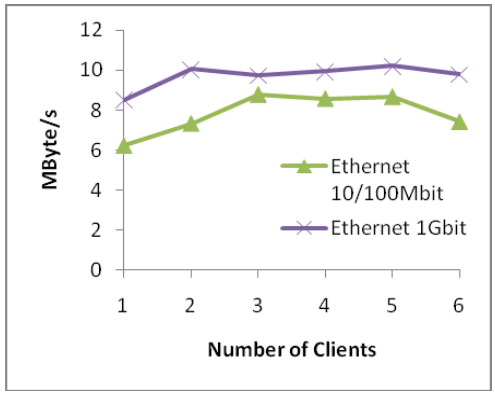

Figure 2: Read throughput under Single Pc Testing, 1 Server, 1 Client PC

Ethernet. The remaining 4 PCs were used to host clients of the file system. Their hardware was 3 x Intel Core 2 Duo E6400 2,13GHz, 2GByte Ram and one Intel Pentium M 1,80GHz – 2GByte Ram, running Windows XP 32bit.

To measure the throughput of each operation supported by our File System we created a benchmark which acts as a File System Client. It is divided into 5 sections: a) create and save 100 files whose size varies from 10Bytes to 10KBytes. Each file is stored in a different path with folder depth varying from 3 to 7. This step was held to test the File System under intensive metadata operations. The folder depth was important because as mentioned to the description of the File System, the absolute path of a file is the key used to store the file's information in the Cassandra. Now since the distribution of the data in the Cassandra nodes is done by those keys, we used multiple folders and subfolders to store our data thus making an even distribution to the Cassandra nodes. b) create 24 directories, again with folder depth varying from 3 to 7. These directories will be used to store larger files and are created separately because writing and reading the big files will result the maximum throughput of the system so in that way we create atomicity to our results (a result corresponds to an operation and only). c) Each client has already created 24 files into its local hard drive whose size is from 512KB to 12MB. At this step client reads the data from 10 of the local files and stores them to one of the directories created from the previous step. This step results the write throughput of the client. d) Read the previously stored files and write them to a local folder (the local folder is different than the one read in step C). This will provide the read throughput. e) delete all the previously created files (100 small files, 10 larger and all the folders and subfolders).

This benchmark as is, needs to synchronize somehow all the clients to execute exactly the same step at any given time. This is why we created a synchronization server. It's a separate program which creates a TCP connection with all the benchmark instances. At startup we define the number of benchmark clients to be expected. Since the only one who can initiate an instruction is the synchronization server, each client at startup connects with the client and waits for instructions. At the time all the expected clients connect to the server, the server instructs all the clients to execute step A. If any of the clients finishes before the other, it waits for a new instruction for the server. When all of them finishes (they all ACK to the instruction of the server), the server instructs all to execute step B and so on. So regardless the number of clients we have and the number of machines we are using, we can be assured that all of them are executing the same operation.

Each client generates a simple text report file. This report includes the overall time of each operation, the overall number of bytes transferred during each operation and the average number of operations per second. The overall throughput of the operations that will be stated next is calculated by summing the total amount of bytes transferred in a single operation from all the clients divided by the operation's time duration of the slowest client. This means that we take as start time when all clients get started and as end point the time when the last client finishes. Overall throughput, as mentioned here, equals with the average throughput of the clients.

Next we will examine the maximum throughput of a single client machine, the effect of consistency configuration to the throughput, the effect of replication configuration to the throughput and finally some statistical data for the metadata operations.

5.1 Single PC Testing

As we mentioned above we used 4 servers running Cassandra 0.7.0 and 4 machines to host clients. First step was to find out the maximum number of clients that is optimal to use on a single PC. For this test we use a single server (remember that all the servers are over GBit) and a single client PC. Through all the testing Write Consistency and Read Consistency both equals to ONE. Figure 1 shows two user cases regarding the write throughput. First line corresponds to clients connected to the same Gbit Ethernet as the server and second line corresponds to clients connected to a simple 10/100Mbit Ethernet. Both cases show that the more clients we call, the more throughput we have.

Over the simple Ethernet 10/100Mbit, one client achieves 7.5 MB/s and tops when using 4 clients to 10.33 MB/s. This is pretty decent because theoretically the network is maxed at 12.5MB/s. If there are 5 clients the number drops to 8.75MB/s so the 4 clients are chosen as the ideal number of clients on a single pc over the simple network.

The testing over the Gbit Ethernet was help for comparison, to see what happens if clients don't have a network limit. At all times, the overall throughput is twice as much. A single client over Gbit is 135% faster (17.8 MB/s) than over a simple network. Also the graph shows a slow increment of throughput event with 6 clients, so a Gbit Ethernet, as expected, can handle more clients at higher speeds.

It is interesting to see Figure 2 which depicts the read throughput of the above clients. It is interesting because the difference between Gbit and simple network is not as much as the one at

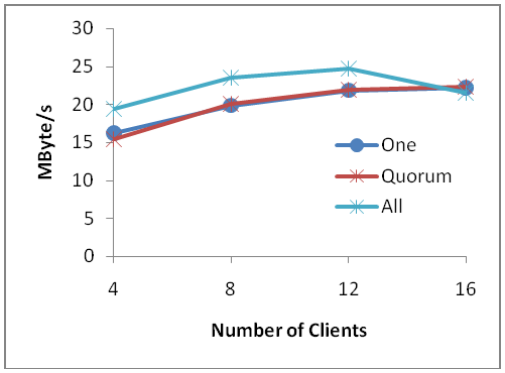

Figure 3: Consistency Level effects to Write speed using 4 servers with Replication 3

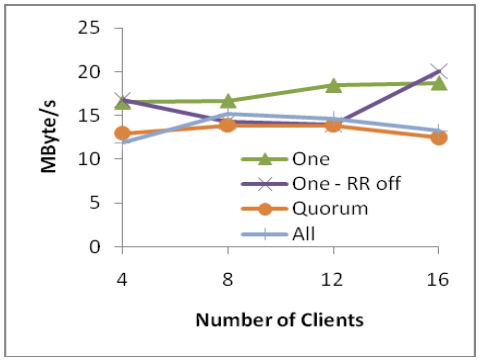

Figure 4: Consistency Level effects to Read speed using 4 servers with replication 3

write operation. 36% increment of speed best shows the issue of slow read speed mentioned to Cassandra configuration. In both cases, using 3 clients and above results to about the same throughput which means that the analogy between time consumption and bytes transferred remains the same and are both increase equally and incrementally.

This first test was essential, because from these results, we ended up using a max number of 4 clients per PC for the rest of our evaluation. At this point you could argue on why we didn't choose the number of 3 clients which is the maximum read throughput considering that read is slower operation. 4 clients were chosen adding the bearable cost of -0.2MB/s hopping that more servers and tuning the consistency level would give us better results.

5.2 The effect of Consistency Level

Next step is to evaluate the effect of Consistency Level configuration to the speed. After this test and on, we stared using all the PCs available. So at this point we have 4 servers and 4 client PCs which will host up to 16 clients. Since we have more than one server we should set up the replication factor. The replication factor was set to 3. Replication 1 wouldn't give us any safety to our data, Replication 2 provides safety giving a spare copy but fearing that a server could come down at any given time and loose our backup we raised the Replication to 3. In this case the possibility that 2 servers are down at a given time is smaller and also in case one server fails there is one spare (hence 4 servers, 3 holding the replica and one of the 3 down). One spare other than the replica holder nodes to keep the Hinted Handoff until the failed server is up again. Replication 4 was rejected because of too much redundant data traveling around.

5.2.1 Write throughput

The Consistency Level is set by each client using the Cassandra servers. So our tests first examined the Write Consistency effect because of the sequence of our benchmark steps. First write, then read. Executing to step C of the benchmark resulted Figure 3.

Figure 3 shows something very interesting and unexpected. Firstly One and Quorum Consistency Level have almost identical throughput. This means that Replication factor 1 or 2 would have exactly the same results because Quorum at a cluster with Replication 3 results to 2 successful writes from 2 servers. Secondly, the unexpected result was the fact that a Write Consistency All is faster than the other configurations. Write consistency All obligates all three serves responsible for storing specific data to ACK, or else the operation will fail.

Most likely the reason why the different configurations didn't have much of a difference was because of the Gbit connection the servers had to communicate each other. So the replicas where instantly broadcasted to the responsible nodes.

We will select Write Consistency All for the rest of the evaluation because of two reasons. First, it's the obvious reason shown in the graph. It's the fastest maxing at 24.7MB/s with 12 clients. Even by serving 16 clients simultaneously it is as fast as the other configurations with 21,54MB/s. The second reason has to do with the Read throughput. As stated before, Cassandra by default has slower Reads than Writes. So the only way to balance this difference was to use Read Consistency One. Upon a successful write operation we are assured that all the nodes responsible to hold a specific replica of data are up to date. If a Read operation takes place immediately after a successful Write operation, by reading any node it will receive the data previously written. This reassurance removes the necessity of checking all the nodes at every read for data consistency leading us to Read Consistency One and theoretically to faster Reads. Of course this decision has implications to the availability of the File System, issue that is not measured in this project at the moment.

5.2.2 Read throughput

Having selected the Write All configuration we can continue by fine tuning the Read Consistency. It is theoretically expected that Consistency One will be the fastest, then Quorum and the All which is proven by Figure 4. The line "One - RRoff" of the graph stands for Consistency level One and Read Repair off. The thought behind disabling the Read Repair was to improve the Read throughput by disabling the unnessasary operations taken that might slower down the operation. Theoreticaly it would be acceptable since the Write All ensures consistency. This thought surprisingly was not proven since at most cases the throughput with Read Repair was faster than the one without. Only when having 16 clients, had the "Read Repair off " slightly better results than the others (20,05MB/s, 7% more than the second one).

As of now, the Read Consistency that will be used will be One.

5.3 The effect of Replication

The number of maximum clients is defined; the Consistency level is set and remains the Replication to be defined. Each Keyspace has its own Replication Factor which is set in the configuration file of the servers along with the definition of the Keyspaces, Families, Columns and Super Columns. All this info is loaded by one server and is broadcasted to the rest and to any other that joins later on. Because the structure of the File System is loaded only once, we first evaluated the FS with Replication 1, closed all the servers, cleaned all their created files and then started a new instance of the cluster with Replication 3 to finish the evaluation.

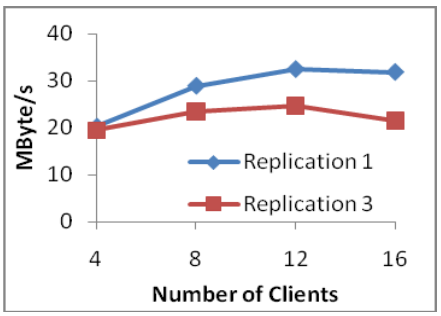

Figure 5: Testing the Write throughput with different Replication factors on a 4 Server Cassandra cluster.

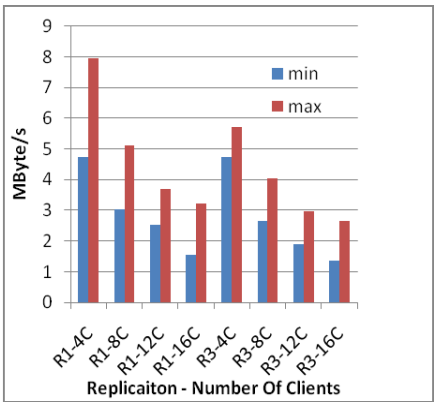

Figure 6: Minimum and Maximum Write throughput of the clients in Figure 5. Horizontal labels format is "Replication X - # of clients".

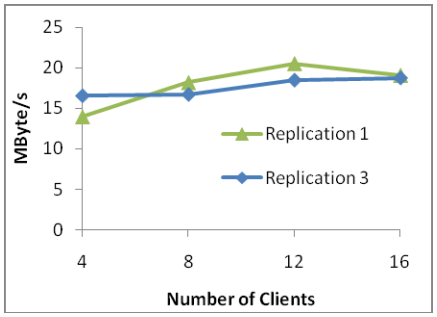

Figure 7: Testing the Read throughtput with different Replication factors.

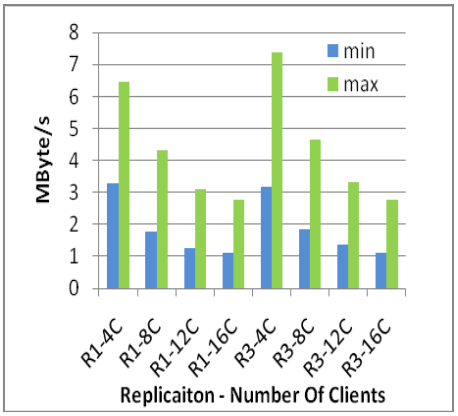

Figure 8: Minimum and Maximum Read throughput of the clients in Figure 7. Horizontal labels format is "Replication X - # of clients".

5.3.1 Write throughput

The results of the benchmark are depicted in Figure 5 and 6. To begin with, if there is one client per PC (4 clients) the two Replication factors are about the same. As the number of clients is increasing, the deference is becoming quite clear. With 8 clients, Replication 1 achieves throughput of 28.9MB/s whereas Replication 3 23,5MB/s (-19%). This difference can be thought as the cost of probing the replicas across the rest of the servers responsible. 12 clients seems to be the ideal with 32,4MB/s write speed at Replication 1 and 24,7MB/s at Replication 3 (-24%). When the number grew up to 16 clients both of the configurations started to get slower.

It is interesting to take a look at Figure 6. This chart shows the minimum and the maximum throughput value of all clients at a given configuration. Here it is clear to see the decrement of a single client's throughput as the number of clients increases. Also this chart shows that it is possible for a client to take more time or even double as much as others so a user of the file system should expect a variance to the time needed for an operation to complete.

5.3.2 Read Throughput

Figure 7 and 8 shows the results of the read operations. From a previous test we decided to use Read Consistency One, a number that doesn't depends on the Replication Factor. So the Read throughput between clusters using Replication factor1 and 3 should be about the same as is in Figure 7. What makes it vary? The more replicas of an object you have, the bigger the chance to find it. This means that a client can get a file from any of the servers holding a replica of that file. It gives the chance to a client to communicate with the closest to it server thus increasing the network speed or to communicate with the least busy server thus it will be served quicker and we will achieve load balancing to our cluster. These two advantages might have more effect to bigger clusters serving more clients.

To return in our case, the maximum throughput is 20,5MB/s at Replication 1 with 12 clients. The interesting part is that when there are 16 clients, if replication is set to 1 the throughput is starting to decrease. On the other hand with replication 3 the throughput is still increasing but with slower rates. This can be considered the advantage in our case.

5.3.3 Metadata intensive operations

Lastly we will take a quick look at the Table 1 which shows the average time per metadata intensive operation. The results are taken by the same benchmark used above in the case of 4 clients. The replication is either set 1 or set to 3. These operations are all effected by Write Consistency level and which is either way set to All.

The metadata, first step of the benchmark. Create 100 files whose size varies from 10bytes to 10Kbytes. We have 4 clients, 100 files each, so 400 files created. Getting the maximum duration of the operations and dividing it by 400 files results an average of 22ms per operation for Replication 1 and 16ms per operation for Replication 3.

The creation of the files, second step of the benchmark. Create 24 paths each of them having folder depth of 3 to 7. Average folder

Table 1: Duration per metadata intensive operation

4C	R1	R3 - All
Small Files	22ms	16ms
make dir	4ms	3ms
rmr	10ms	7ms

depth is considered the number 5. So 24 paths X 5 mkdir operations equals to 120 operations per client. We have 4 clients and overally 500 mkdir operations. Again, we extract the maximum duration of this step from the for clients and we divide it by 500 operations. This equals to 4ms / op for Replication 1 and 3ms for Replication 3.

The deletion of all the files created. Each client has to delete 120 folders and 110 files (100 small and 10 big files). Again we divide the maximum duration of this step by 120 folders plus 110 files. This results an average of 10ms / op with Replication 1 and 7ms with Replication 3.

All three operations where faster with Replication 3 than with Replication 1 something that was surely not expected.

5.4 General thoughts

The evaluation was proven quite useful, because it gave us information on things that we could only imagine. All the process to run the benchmarks, gave us the opportunity to find the right configuration of the Cassandra servers because during the test it was natural to have our fails, exceptions, failed servers and all. Every failed server gave a small part of information on something that went wrong. This kind of fine tuning will continue for a long time.

Now since we finished our evaluation, it is interesting to note that the best performance was indeed achieved when using 3 clients per PC. You might remember, at the beginning of the evaluation, during the Single Pc Testing, we saw peeks at the graphs at the positions where 3 clients were used. At that time we chose 4 clients as maximum hopping to see better results with other configurations but it turned out it didn't. So that simple one to one benchmark provided us information that was proven correct by the rest of the evaluation.

One thing that wasn't examined is the effect of the chunk size in the File System to the overall throughput. Generally speaking, the size of the chunk size should be set according to the average size of the files that will be stored in the File System. There are a few things someone has to be careful with when changing that value. Increasing it too much might exceed the maximum Thrift message size (the basic messages exchanged between Cassandra clients and Cassandra servers). Also bigger chuck sizes needs more available Ram memory, because when the compaction is executed, rows are loaded into memory. Increasing the row overall size increases the requirement for memory.

Lastly, a statistical information that was interesting; during all the benchmarks the clients read and wrote 62GByte of data. The servers had higher network traffic because of the replication. One of the servers that was monitored had read and written 210GByte of data (yeah, that much redundant network traffic for data safety).

6. CONCLUSION

We saw how a Database-like system can be successfully used to implement a File System API. Especially if that system is Cassandra we inherit all its cool features like scalability, availability and fault tolerance. Nowadays using open-source projects we have the ability of creating a Highly Scalable File System using an already tested system with minimal cost.

7. REFERENCES

[1] A. Lakshman., P. Malik, 2009, Cassandra - A Decentralized Structured Storage System. In ACM.

[2] Giuseppe de Candia, Deniz Hastorun, Madan

Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon~Os highly available key-value store. In Proceedings of twenty-_rst ACM SIGOPS symposium on Operating systems principles, pages 205{220. ACM, 2007.

[3] Fay Chang, Je_rey Dean, Sanjay Ghemawat,

Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7

[4] cassandra-fs.

http://code.google.com/p/cassandra-fs/

[5] Hector client for Cassandra.

http://prettyprint.me/2010/02/23/hector-a-java-cassandra-client/

[6] Apache Thrift.

http://thrift.apache.org/