Reliability of Data-Intensive Distributed File System: A Simulation Approach

Corentin Debains*^, Pedro Alvarez-Tabio Togores*◦, Firat Karakusoglu*

*Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA
^Telecommunications, ENSEIRB-MATMECA, Bordeaux, FRANCE
◦ETSI Telecomunicacion, Universidad Politecnica de Madrid, Madrid, SPAIN

ABSTRACT
The data deluge of technical computing has driven the emergence of data-intensive computing paradigm such as MapReduce. Data-intensive distributed file systems are core component to cooperate with runtime systems to support data-intensive applications. Fault tolerance is one of the primary design goals for data-intensive distributed file systems due to the assumption of commodity hardware. Hadoop file system (HDFS) is one of the most widely used DI-DFS systems and its reliability can change based on different factors including replication degree, heart beat poll interval, network bandwidth and block size. In this project, we adopt a simulation approach to systematically understanding the reliability of HDFS based on defined durability metric. The resulting simulator measures the system's durability by running the file system according to values given in configuration file that affect reliability while introducing random node failures to the system. Simulation allows us to quantify the effect of these factors without setting hundreds of thousands of nodes in real, and thus can serve as a tool for analyzing required settings to achieve a defined durability level. This report outlines our research and implementation and evaluates the performance of the Hadoop file system reliability algorithm.

Categories and Subject Descriptors
D.4.5. [OPERATING SYSTEMS]: Software, Operating Systems, Reliability
D.4.8. [OPERATING SYSTEMS]: Software, Operating Systems, Performance – Measurements, Modeling and prediction, Monitors, Operational analysis, Queueing theory, Simulation, Stochastic analysis.

Conference'10, Month 1–2, 2010, City, State, Country.

General Terms
Algorithms, Measurement, Performance, Reliability, Experimentation

Keywords
replication, simulation, distributed file system, monitoring, Hadoop, HDFS

1. INTRODUCTION AND MOTIVATION
Data-intensive distributed file systems (DI-DFS) are developed as the file systems to support data-intensive computing frameworks such as MapReduce [*1*]. Data-intensive file systems break down the data into chunks/blocks and distribute them onto each node with the consideration of data locality. Replication based data placement is utilized by data-intensive file systems to tolerate failures. Leading data-intensive file systems include Google File system (GFS) [*2*], Hadoop File system (HDFS) [*3*] and CloudStore [*4*]. The deployment of data-intensive file systems calls for local attached disk on each compute node to foster scalability and data locality, which is projected to be the trend for future HPC systems [*5*].

Reliability is one of the primary design goals for DI-DFS, due to the fact that target platforms of data-intensive frameworks are based on commodity hardware. The problem is amplified for large-scale data-intensive applications with hundreds of thousands of compute nodes with more prevailed failures. Note that hardware/software failures are not the only sources to threaten the reliability of DI-DFS. Any other factors that cause the interruption of the services can hurt the reliability. For example, some research proposes to power off some data nodes in DI-DFS to save energy, which can be considered as another factor to impact the reliability [*6*].

Replication is the state-of-art approach to handle failures for data-intensive distributed file systems. In particular, DI-DFS adopts multiple replicas for each data block. Whenever the replication degree for a data block drops below the configured amount, the

namenode forces the block to be re-replicated on the remaining data nodes [4].

However, no experiment was made to compare the replication algorithm and implementation on DI-DFS. It is impossible to compare different systems based on their reliability through failures. The most common reliability performance comparison has mostly been against RAID systems [11] which were defined 23 years ago and provide a hardware level replication and distribution. But the reliability achieved by replication is not implemented as a dedicated reliability system but integrated into the system architecture and block management, making it highly customizable and easier to manage.
Moreover, such reliability systems didn't see their performances measured as the other parts of the system and are based on on-site experiments and development assumptions, like in [2]. We wanted in this project to provide an analysis and experiment of this kind of system and evaluate their performance in a environment when the worse scenarios can be imagined, for example massive byzantine failure or central network issues, which could make the system unstable or, worse, cause a loss of data. Our main concern was to measure the performance in such environments and explore the limits that this kind of system can face, mostly in terms of scalability. As the cluster size that will be used by such systems are likely to go to a million nodes in the next few years [5], the scalability of the replication process could be a limitation for the whole system scalability.

This paper is organized with the following structure: section 2 draws a general background to our project, section 3 explores different related work we used to base our research, section 4 defines the solutions we chose to implement, section 5 describes our implementation, experiments and results, and section 6 gives conclutions and describes possible future work.

2. BACKGROUND

DI-DFS use a centralized component called Namenode and several storage nodes called Datanodes. The Namenode handles the metadata and block locality, so is responsible for the replication maintenance of blocks whereas Datanodes store the blocks, transfer data with clients and report activity and issues to the Namenode.

Whenever the replication degree for a data block drops below the configured amount, the Namenode forces the block to be re-replicated on the remaining data nodes [4].

The reliability of DI-DFS is impacted by multiple factors, which are listed as follows.

The number of replicas (Replication Degree). Obviously, the reliability is proportionally impacted by the replication degree. A higher replication degree usually means higher reliability. However, the storage space is not free, and more data communication cost will be incurred for more replicas. There is a trade-off with the selection of replication degree: we need to balance between reliability and space utilization.

Heart beat poll interval. The datanode periodically informs the namenode that it is alive and poll the datanode for commands to process. A shorter poll interval helps to identify the corrupted nodes and data blocks earlier and transmit more often new commands to process if necessary. However, the shorter poll interval will inevitably burden the centralized metadata server.

Network bandwidth. The re-replication depends largely on the network bandwidth in the system. The problem is exaggerated for distributed computing environment, where the upload network bandwidth is bounded with several hundreds of Kb/s only.

Block Size. Intuitively, block size impacts the reliability of DI-DFS. However, the impact is still vague. A larger block size usually implies more cost in re-replication. However, the number of corrupted data blocks will also be reduced with higher block size.

The data placement mechanism. First, if the failure patterns of the Datanodes are correlated, an intelligent data placement mechanism is needed to disperse the replicas of one data blocks to unrelated nodes. Furthermore, even if the nodes are not correlated, the placement of data blocks for the same file could still be optimization problems.
Simulation is also commonly used to evaluate the performances of distributed systems without using real networks and infrastructures which can be costly and hard to manage. In [13], they had a similar approach to evaluate reliability of a system without deploying it.

3. RELATED WORK

In [2], a rudimentary measure of performance of the replication system is made on a cluster using GoogleFS. As a close implementation, HDFS should have close behavior. They killed a node in a 227 nodes cluster an observed the replication of about 15000 chunks in 23 minutes with their parameters, which were maximum 91 concurrent replication transfers and a maximum bandwith of 6.25 MB/s. They also tried another experiment with a simultaneous provoked failure of two different datanodes with each 16000 chunks. This provoked 266 chunks to get only one replica left instead of three and so triggered with the highest priority. They evaluated that it took 2minutes before those 266 chunks get at least another replica.

In [7], the authors extended Hadoop into volunteer computing environment. In particular, they proposed dedicated computing resources to make up the

unavailability of MapReduce due to the vulnerability of each node in volunteer computing environment. This work brings the attention to the unavailability issue of MapReduce framework. However, this work focuses on the impact of failures on MapReduce application performance, i.e., application elapsed time, not the reliability of file systems.

MRPerf is a MapReduce simulator tool to model the application performance on large clusters [8]. MRPerf was developed based on ns2, and focuses more on the application performance and the impact of network bandwidth. The reliability of DI-DFS, however, is not the objective of MRPerf.

In [9], the authors presented an empirical study on the impact of failures on the Hadoop applications. Again, the file system reliability is not concerned in this study.

In our first approach for a simulation, we decided to look into the ThriftStore simulator code, presented in [13], in order to get a first-hand knowledge and inspire the structure of our simulator, or even adapting the ThriftStore simulator to our project. ThriftStore is a storage architecture that integrates two types of components: volatile, aggregated storage and dedicated, yet low-bandwidth durable storage. On the one hand, the durable storage forms a back end that enables the system to restore the data the volatile nodes may lose. On the other hand, the volatile nodes provide a high-throughput front-end.

4. PROPOSED SOLUTION

4.1 Define a Metric: Durability

The first part of our project is to define metrics that will be measured by the simulator. In order to have a better idea of what could be used; we extended our readings to reliability oriented papers to get a better overview on how other researchers evaluated the reliability offered by their systems and what metrics have been used.

So each of us read a set of papers and reported to the others what kind of reliability the system offers and how the authors evaluated the performance of those systems. We merged our results and observations to define our own metrics which could be as much global as possible and compatible with the HDFS reliability system.

A lot of research has been carried out to enhance replication in distributed file systems in order to achieve the desired availability and above all reliability of data in a distributed computing environment. Starting in [2], the authors present the data replication in the Google File System. The replication order is 3 by default. The metrics they used are of great interest: they measured the mean time to repair for different levels of chunks (emergency replication and rebuilding for failure replication).

In [10], the authors propose a QoS-based approach to storage. QoS is a well-known term applied mainly to computer networking. Examples such as Frame Relay, DiffServ or IntServ have enjoyed great acceptance in the community. The basic concept is to combine storage caching with replication. The storage QoS specification comprises two concepts: traffic profile and performance requirements. The latter specifies metrics, some of them related with our work, such as acceptable miss rate or availability/redundancy. It is not directly related with our project but we could use some of the concepts explained (especially caching combined with replication) to improve our simulation and possible solutions to the problem of re-replication.

The benchmarking of a programmable environment for distributed storage is also explored in BitDew [14]. However, BitDew is not strictly a distributed storage as it only has file-level storage and replication, not block-level. Moreover, BitDew uses DHTs to handle the file references while common distributed file systems, such as Google File System, use a centralized approach. We find the Data Scheduler algorithm presented really interesting, as it handles replication with only setting of two parameters (called fault-tolerant and replicas). This may help us developing different solutions to data replication.

In [12], the authors present a possible solution to the problem of fault-tolerance in cooperative storage. Their main assumption is that creating massive redundancy will ensure a low probability of data loss even considering large-scale correlated failures. The most important concept for us is durability, which is the probability for at least one file replica to survive after a large-scale correlated failure.

Finally, an important starting point from our next steps regarding simulation and metrics definitions is [13], especially those related with durability and its cost, and the tradeoffs between availability and durability.

These related works helped us to define three different metrics which are described as follow.

Durability. The main metric that we are going to measure is Durability. Durability means the probability that a file has to survive to a specific large-scale system failure, such as what was performed in [12]. A durability of 0.99 means that the file has 99% probability to still being available after the introduction of the failure. The durability will depend on the parameters we chose in our proposal but also on the type of failure or storage efficiency.

The type of failure is a simulation parameter, which could be a large-scale byzantine failure or a large-scale scheduled failure, giving different durability for instant global failure and growing failure. The storage

efficiency is related to the system parameters and will allow comparing the durability with other systems.

Mean Time to Repair. This sub-metric is for knowing the time required to repair the system once failures began to happen. During implementation, we will count the time in the meaning of seconds once system starts repairing the system till it reaches a full reliable state. The repair process can be measured to different steps, the emergency repair and total repair, similarly to the metrics used by Google in GFS [2].

Workload & Scale. The workload that we are going to submit to the system should be enough to fill around 30% of the system storage and the simulator will keep submitting data over time to keep activity into the system. The scale will be linked to the simulator scalability and we don't want to corrupt the observation with simulation limits. So we plan to measure the simulator's performances to adapt the simulated scalability. We expect to be able to scale to thousand nodes.

4.2 Design a Simulator

In Thrift Simulator, two types of nodes are simulated: volatile and durable (different replica placement policies for each of them). Both of them have limited bandwidth, but durable has unlimited storage, in contrast to volatile, which doesn't. As HDFS doesn't sort nodes by reliability during replica placement, we chose not to implement this feature inside our simulator.

The simulator is driven by a trace of failure events (transient and permanent) for the volatile nodes. Also the simulator monitors the state of the objects (replication level), and the amount of traffic generated by all nodes throughout the simulation. Its parameters by default are, among others: Bandwidths, Replication levels, Replica placement algorithm, Failure rates, Node count. However, the parser of program arguments is able to modify defaults by passing custom parameters. We used that approach by designing a program able to create a schedule of failure and made the simulator. As a tool to evaluate the properties of HDFS and its behavior during major failure, it needs to implement a measuring system which will give results to the user on the simulation; it should also process the results and serve the metrics we defined. In order to fully control the failure introduction, the simulator should respect the failure schedule which would have been previously generated to make the failure distribution follow some well known scenario: exponential, linear, poisson…

From our study of ThriftStore, we can extract some useful ideas and functions. For example, we can identify durable nodes in ThriftStore with central metadata management in Hadoop (NameNodes). Another very interesting concept we took for our simulator is the MetadataService implementation in ThriftStore simulator. Another important point is that the simulator is intended for the data replication part of ThriftStore, it has things in common with Hadoop but it does not implement any Hadoop code for replication, so in case we adapted it, we would have needed to implement it as an add-on to Hadoop. This can be a serious problem, as the difference in languages of implementation (Python vs Java) adds to the inherent differences between ThriftStore and Hadoop.

For this reason, we concluded that it is way easier to translate the simulator to Java and integrate Hadoop into it than to integrate Hadoop replication in Python code.

Our approach to evaluate the performance of the replication system integrated to HDFS was to build from scratch a simulator which will focus on this system and simulate the worst environment for HDFS to push it to its limits.

Building a simulator involves two different issues that we had to deal with at anytime: simplify the system to make it easily manageable and focused on the studied part but also make it as close as possible to the original system to keep close to the reality and get probant results. Simplify HDFS meant to produce a system able to run on a single host or few computers, simulate a big scale deployment and abstract some parts to avoid complexity. Keep it close to reality meant that we had to be very respectful of the official implementation, understand it and appreciate the importance of every feature and its effect on our measurements. The tradeoff between the two constraints was one of the main aspects of the design. Our analysis of the HDFS replication system is described in the next section, while we want to focus on the simulator architecture.

Our first assumption was that we start with a balanced distribution respecting the chosen parameters, which was built as a static image of a HDFS system, with a namenode, several datanodes and blocks distributed on the Datanodes and known by the namenode. This image would be the starting point of the simulation, when the failures are going to be triggered and the reproduced replication system will handle the failures.

Our second assumption was that we wanted to abstract the network by keeping the whole simulator on the same machine. This assumption would allow having a perfect or custom network and not relying on a real network which could show limitations in the simulator because of the scale the simulator has to achieve.

Also, we had to keep a parallelism inside the simulator to hope sticking with the original HDFS implementation which runs several thread at the same time either in the Datanodes and the namenode.

Therefore, the simulator will be a multi-thread program.

The purpose of the simulator is to evaluate the performances of HDFS for different parameters which are tweakable but also scale the system up to see its performance and resistance to failure on a large node distribution. The simulator's design should be scalable to at least 100000 Datanodes.

Finally, one of the main concerns is the accuracy of the simulator. The simulator is meaningful only if it has the same behavior as the real HDFS system, so we will have to compare both real and simulated behavior.

5. EVALUATION

5.1 Durability in HDFS

In HDFS, data loss may happen because of either a hardware, software or environmental failure. We won't go into details about software and environmental failures of the system, as our main concern is hardware failures and how it is handled by replication mechanism. Multiple hardware components may fail simultaneously or some corruption may occur on disks that may possibly cause data loss. Those type of hardware failures are observed by replication system.

Data node failure is one example of hardware failures that name node should detect after not receiving related data node's heart beats for some period of time, 10 minutes in default [15]. When a particular datanode is detected as dead, its replicas are re-replicated on other nodes to keep the number of replicas in defined value.

If any corrupt data is detected, then system behaves differently and checksums [16] are used to check data's validity. Hadoop verifies data nodes on the receipt node, if checksums don't match, then related data block is sent back. Every read and write operation requires checksums, if any are found then name node is reported and re-replication process is started.

The stored data in HDFS is not read quite often but when it is required to read it should be avaliable without any problems. To achieve this kind of availability, each data node runs a background thread to check data blocks integrity. If any corrupted blocks are found, then namenode is informed and re-replication process is started for corrupted data blocks. Data block scanner checks disks against disk errors every three weeks. [17]

5.2 Replication System in HDFS

The Replication System in HDFS is located exclusively in the namenode, which is the only member to have an overview of the whole system and so the ability to decide if a block has to be re-replicated and the status of all datanodes. The replication system consists of few different parts which all have their own workflow for optimization purpose. We identified 3 different workflows which participate to the reliability of the system.

HeartBeat Monitor. This part is periodically checking, by default every 300s, the list of Datanodes and especially the last time they sent a proof of activity: the hearbeat. If the last time they sent the heartbeat is older than a specific time, the datanode is marked at unavailable and removed from the list of active nodes. We identified the allowed interval between two heartbeats as being by default the interval of 2 recheck interval plus 10 heartbeat interval (3sec), so 630s by default, which means a datanode can skip some heartbeat without being marked as failed. This can happen if the datanode experience a slow down which delay its heartbeat or if the network has intermittent problem. The parameter is voluntarily high to avoid extra replication and so a replication storm which could slow down the system because of the network overuse but, on the other hand, it may not be very reactive in case of a byzantine failure where the system fall gradually. When a node is marked as unavailable, all the blocks that were stored on it see their number of live replicas decreased of one unit. If they were not over-replicated, which sometimes happen after the return of an ex-unavailable node, the block is scheduled to be replicated by being added to a list called *neededReplications*.

Replication Monitor. This part monitors the list we filled in the heartbeat monitor, neededReplications. Periodically, every 3 seconds by default, the namenode checks this list by order of priority and generates replication jobs that will be attributed to datanodes. The priority is based on the number of current replicas available on the system. There are 3 different levels: the highest priority is for blocks where only one replica is available or if the replicas are only store by node that are going to be shutted down (decommissioned), meaning that replication is critical and a failure could mean a data loss. Then medium priority is for blocks where the number of live replicas is only the third of the required number of replicas, finally, the last level is for all the blocks that are under replicated. The selected blocks, a limited number to avoid network congestion, are scheduled for replication. To create a replication job, the namenode has to find a source and a target to allow the replication and a good data placement. The source is chosen among the blocks containing nodes with some discrimination considerations (current transfers in progress, decomissionned status, block already requested to be deleted or node corrupted) and a random factor. The target choice is made exactly the same as the data placement made when distributing the block for the first time: it may be placed on the same rack, in another close rack and if those conditions are already filled, randomly on an available node. The created job is added to the chosen source node list of

jobs that will be given to it in answer to its next heartbeat. Parallely, the triple is added to the pendingReplication list to keep a trace of the pending replication jobs.

Also, the Replication Monitor checks the marked, by the pending Replication Monitor (described below), timed-out replications jobs and re-add them to the *neededReplication* list in order to get them processed again for replication.

Pending Replication Monitor. The list called *pendingReplication* is used to check that the replication jobs are well executed by the nodes and that a block is correctly replicated. Periodically, every 5 minuts by default, the pendingReplication list is checked and timed-out jobs are considered as failed and marked as timed-out to be processed again. A pending replication failure can be caused by several reasons, the target or source node became unavailable after the the creation of the replication job, an error occurred in the transfer or the network is too slow between the two nodes and the replication has not been processed yet.

5.3 Simulator Implementation

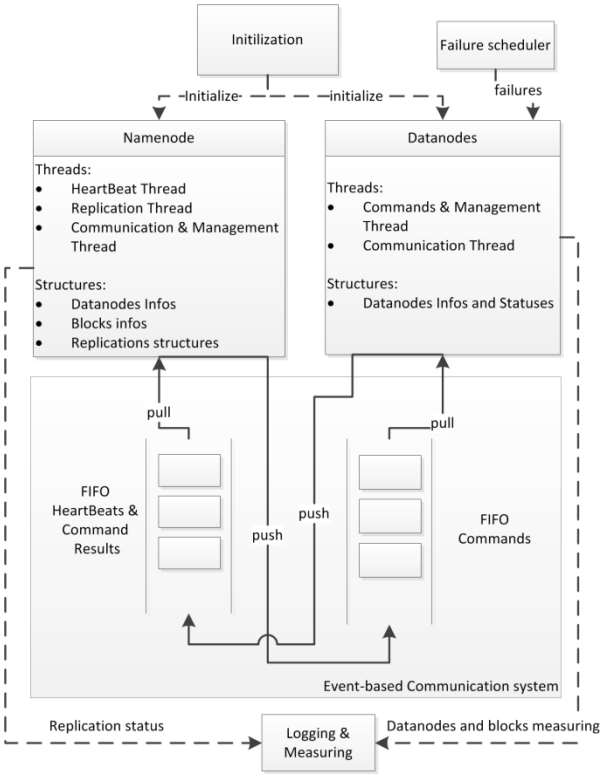

Figure 1 - Simulator Architecture

We built the simulator around two main structures defined next, a namenode and a datanode manager called AllDatanodes. The architecture is described in Figure 1 and explained below.

Namenode. We identified the Namenode in HDFS as the responsible for a great part of the replication monitoring and tasks, so we tried to keep the structure close to the original implementation. The namenode in our simulator was designed as a structure with different threads to achieve all the tasks that we kept, those relative to the replication system. We kept three different threads that are running and HDFS and we reproduced their actions.

Hearbeat Thread. This thread check the heartbeat table containing the heartbeats sent by the datanodes to say that they are alive. If a datanode has not sent a heartbeat within a specific time since its last heartbeat, it is considered as dead and so is remove from the active node list and the blocks that is was storing are scheduled for re-replication to repair the lost replica.

Replica Thread. This thread checks the list of scheduled for replication blocks, order them by priority and then prepare for the replication. It chooses a source node among those still holding the block, choose a target among the nodes and schedule this triple to be attributed to the source node for processing at his next heartbeat push/pull.

PendingReplicationThread. This thread checks the list of scheduled triple and re-schedule the block to be replicated if the triple request has timed-out.

AllDatanodes. The AllDatanodes structure is responsible for all the datanodes because the simulator does not run a single datanode individually but let the AllDatanodes structure manage all the storage nodes. We chose to abstract the concept of datanode because we evaluated that the namenode could be the bottleneck for the replication system and that the datanodes does not represent an interesting system themselves to be run individually. So we simulate a bunch of nodes with a single object. This object keeps track of all the datanodes informations in a table with their state, their capacity, their blocks, and their bandwith use. AllDatanodes run two threads to manage all the nodes such as the Heartbeat system which determines when for each datanode if it has to send a heartbeat to the namenode to make it think that this datanode is alive. The second thread is made to process the commands transmitted by the namenode through the communication system integrated in the simulator.

Communication System. The simulator is not using regular RPC communication as in the regular HDFS system but a simplified system allowed by the single-host deployment. Two structures are global on all the system, called toDatanodes and toNamenodes. These structures are composed of events that are sent either to the Namenode or to a Datanode. To the namenode, it can be a heartbeat or a block reception acknowledgment whereas to the datanode, it can be a replication command to send to a datanode. Both

AllDatanodes and Namenode have a dedicated thread to process the events queued.

Failure introduction and Measure. Above the previous objects, we created a failure introducer which respects a schedule of failure which has been previously created by another program we made. Then, a logging structure enables to store action and timestamp to know exactly what happened and the system and be able to draw an evolution and give results.

5.4 Testing Environment

We have tested our simulator in two quad-core PCs. First, we had to generate a simulation file for each setting using the SimFileGenerator we have developed. This file generator defines the parameters for the system as well as the scheduled failing times. We have considered worst cases of extreme failing in the nodes (not those of reality) and have increased the bandwidth between nodes accordingly to shorten our simulation time and be able to get results more quickly, especially regarding durability comparisons. A typical configuration file is generated with these arguments:

Replica is 5, bandwith is 32768 kbit/s, block is 64MB, heartbeat is every second, MTTF is 3s, distribution is poisson, nodes is 500, failPercent is 40% and number of blocks is 100000.

Equation 1 - Poisson Distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!},$$

Equation 2 - Exponential Distribution

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0, \\ 0, & x < 0. \end{cases}$$

We used a Poisson distribution, equation 1, to model the real failures given expected failures as a parameter. The expected failures are just nodes*failPercent/100. In this particular case, the simulation will end when 40% of the nodes are simultaneously unavailable. For the failure arrivals, we have defined an exponential random variable, equation 2, with rate equal to Mean Time To Failure (MTTF).Here we have inspired from the queueing and traffic theory [19]. This theory describes the time between events in a Poisson process, i.e. a process in which events occur continuously and independently at a constant average rate. A sample file introduces the same parameters as those given to the scheduler and add schedule described by {nodeId,timestamp} pairs.

Given this file, the simulator will read them and start scheduling failures given these parameters.

5.5 Performance Evaluation

Tradeoffs between replication and Durability. The objective of the first set of simulations was to find out the relationship between Durability and Replication under a massive byzantive failure of 40% distributed over time as as exponential distribution. We carried out 10 experiments for each set of values, and then get the mean of those results. The setting (constant values) were the following. The replication level went from 2 to 10, and our objective was to see if we could get approximate values to a commercial implementation of a reliable data store. Amazon S3 [18] claims to have a durability of 99.999999999%. Due to the limitations of our simulator (it starts crashing when replication level is too high), we tried to achieve a more modest result of 99.9999%. We achieved it with a replication level of 6. We plot the results from replica=2 to replica=10, on figure 2, to see the tendency more clearly.

System Load, Block number vs MTTR. In our second simulation the main objective was to find out how the number of blocks in the system overloads it with the same kind of failure. As we defined Mean Time To Repair, we thought it could be interesting to plot the variations in MTTR while injecting more blocks into the system. We also want to compare what is the tradeoff between replication level and MTTR.

We can clearly see how a replication level of 8 affects MTTR in figure 3. This is because, as the number of replicas increase, so does the number of blocks in a node. When this node fails, we have to replicate more blocks than with less replication. This is considering

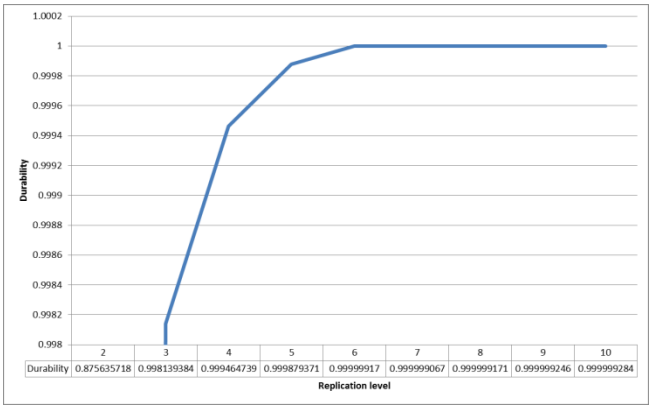

Figure 2 - Durability related to number of replicas under major failure

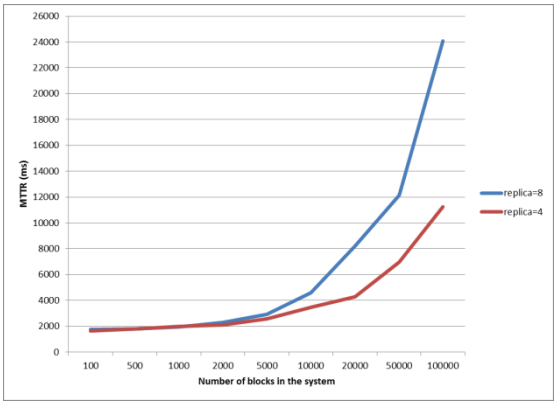

Figure 3 - MTTR related to number of blocks

that we do not reach the maximum capacity of the node, which is the case. This difference in MTTR between 4 replicas and 8 replicas can be significant when reaching a great load in the system. The results may not be the same as a real deployment, but they can show us a tendency in how MTTR is affected.

5.6 Evaluation Limits

Even though a real implementation of HDFS is supposed to handle up to 10,000 nodes, our simulator is able to handle a greater number, as it starts being unstable beyond 50000 nodes and few millions blocks. This is surely an implementation issue, but also a hardware limitation: maybe we need a more powerful machine to carry on the simulation in a real scale.

Another issue is with the design of AllDatanode. In our simulator, to improve the throughput and performance in a single machine, we stick to a single-threaded Datanode manager, which can provoke a performance bottleneck in a real distributed environment.

Finally, as the simulator only comprises the replication part of HDFS, it would be good to extend it and add functionality to mirror more closely the HDFS behavior.

6. CONCLUSION

In this project, we have achieved a great knowledge of the internal replication mechanisms in HDFS, which are not at all trivial. First we conducted a thorough research of all the replication mechanisms behind many distributed systems, focusing after in HDFS. Our goal behind this research was to set the foundations of an HDFS replication simulator as close to reality as possible. With this simulator, we are able to find out the relationship between durability, Mean Time To Repair and important parameters in HDFS such as the replication level, the heartbeat interval, or the number of blocks in the system.

The work was divided as fairly and equally as we could among us. The simulator design comprises the contributions of all three os us. Part of the theoretical foundations were devised by Firat. Corentin read the Glacier paper and defined the durability and the concept of byzantine failures. Pedro started the research about the simulator by taking some interesting concepts from the ThriftStore simulator. Finally, Corentin and Pedro implemented the simulator and all of us carried out the tests. We also have to acknowledge Hui Jin for his invaluable advices and guidance throughout this project.

The HDFS simulator we developed represents an accurate tradeoff between simplification and the actual behavior of HDFS. Our results show that great durability in HDFS can be achieved with a relatively low level of replication. We can achieve a 99.9999% durability with only a 6-level replication, which is a really nice value, but still a bit far from Amazon's S3 specifications. On the other hand, we have found out that MTTR is very dependent on the load in the system (number of blocks present), and with high replication levels it can scale up dramatically.

Future Work. The main lines of our future work are basically summarized in extending the simulator towards getting more accurate, closer to reality results. For example, we can support load balancing as HDFS specifies. Another possible improvement would be to handle corruptions in data blocks. We haven't contemplated it, but it is another source of data loss that affects durability and other metrics we defined. Finally, adding checksums to Datanodes would be closer to the real implementation. In the actual HDFS it is done once every ten days in the default configuration.

7. REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM - 50th anniversary issue: 1958-2008*, vol. 51, no. 1, 2008.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung., "The Google File System," in *Proc. of ACM Symposium on Operating System Principles, SOSP*, 2003.

[3] Hadoop Distribute Filesystem Website. [Online]. http://hadoop.apache.org/hdfs/

[4] Kosmos Distributed Filesystem (CloudStore) website. [Online]. http://code.google.com/p/kosmosfs/

[5] I. Raicu, I. Foster, P. Beckman, "Making a Case for Distributed File Systems at Exascale," in *Proc. of ACM Workshop on Large-scale System and Application Performance (LSAP)*, 2011.

[6] Kaushik, R.T.; Bhandarkar, M.; Nahrstedt, K., "Evaluation and Analysis of GreenHDFS: A Self-Adaptive, Energy-Conserving Variant of the Hadoop Distributed File System," in *Proc. of Second International Conference on Cloud Computing Technology and Science (CloudCom)*, 2010.

[7] H. Lin, X. Ma, J. Archuleta, W. Feng, M. Gardner, Z. Zhang, "MOON: MapReduce On Opporutunistic eNvironments," in *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[8] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta, "A Simulation Approach to Evaluating Design Decisions in MapReduce Setups," in *Proc. of the 17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer*

and Telecommunication Systems (MASCOTS), 2009.

[9] H. Jin, X. Yang, X. -H. Sun and I. Raicu, "An Empirical Evaluation of MapReduce under Interruptions," in *Cloud Computing and its Applications (CCA)* , 2011.

[10] J. Chuang, M. Sirbu. "Distributed Network Storage with QoS guarantees". Chuang, Sirbu. Carnegie Mellon University, University of California Berkeley.

[11] David A Patterson, Garth Gibson, and Randy H Katz, ACM 1988, A case for redundant arrays of inexpensive disks (RAID). ACM, New York, NY. DOI = http://dx.doi.org/10.1145/50202.50214

[12] A. Haeberlen, A. Mislove and P. Druschel, "Glacier: Highly durable, decentralized storage despite massive correlated failures", *2nd Symposium on Networked Systems Design and Implementation*, Boston MA, May 2005.

[13] A. Gharaibeh, S. Al-Kiswany and M. Ripeanu"ThriftStore: Finessing Reliability Trade-Offs in Replicated Storage Systems", *IEEE Transactions on Parallel and Distributed Systems*,2011.

[14] G. Fedak, H. He, F. Cappello. "BitDew: A Programmable Environment for Large-Scale Data Management and Distribution". INRIA.

[15] Hadoop Distribute File System. Default Configuration Description. http://hadoop.apache.org/common/docs/r0.20.0/hdfs-default.html

[16] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, The Hadoop Distributed File System, 978-1-4244-7153-9/10/$26.00 ©2010 IEEE

[17] MapReduce for the Cloud - HADOOP The Definitive Guide - Tom White - ISBN: 978-0-596-52197-4 page:283

[18] Amazon Web Service. Simple Storage Service (S3). Website. http://aws.amazon.com/s3/faqs/#How_durable_is_Amazon_S3

[19] Shlomo Halfin and Ward Whitt – Heavy-Traffic Limits for Queues with Many Exponential Servers.