2012 31st International Symposium on Reliable Distributed Systems

From Backup to Hot Standby: High Availability for HDFS

André Oriani and Islene C. Garcia
Institute of Computing
University of Campinas - Unicamp
Campinas, São Paulo - Brazil
Email: andre.oriani@students.ic.unicamp.br, islene@ic.unicamp.br

***Abstract*—Cluster-based distributed file systems generally have a single master to service clients and manage the namespace. Although simple and efficient, that design compromises availability, because the failure of the master takes the entire system down. Before version 2.0.0-alpha, the Hadoop Distributed File System (HDFS) — an open-source storage, widely used by applications that operate over large datasets, such as MapReduce; and for which an uptime of 24x7 is becoming essential — was an example of such systems. Given that scenario, this paper proposes a hot standby for the master of HDFS achieved by (i) extending the master's state replication performed by its checkpointer helper, the Backup Node; and by (ii) introducing an automatic failover mechanism. The step (i) took advantage of the message duplication technique developed by other high availability solution for HDFS named AvatarNodes. The step (ii) employed another Hadoop software: ZooKeeper, a distributed coordination service. That approach resulted in small code changes, 1373 lines, not requiring external components to the Hadoop project. Thus, easing the maintenance and deployment of the file system. Compared to HDFS 0.21, tests showed that both in loads dominated by metadata operations or I/O operations, the reduction of data throughput is no more than 15% on average, and the time to switch the hot standby to active is less than 100 ms. Those results demonstrate the applicability of our solution to real systems. We also present related work on high availability for other file systems and HDFS, including the official solution, recently included in HDFS 2.0.0-alpha.**

I. Introduction

Hadoop Distributed File System (HDFS) [1] is a very reliable and scalable storage system. It provides data with high consistency, throughput, and availability. It is capable of storing about 21 PB [2] distributed among 4,000 nodes, and serving 60 million files to up to 15,000 clients at data throughputs higher than 40 MB/s [3]. Inspired by the design of Google File System [4], HDFS was initially developed at Yahoo to store web crawling indexes. Nowadays, it is a core component of Apache Hadoop [5], providing storage service to the other components of the project. Those components include Hadoop MapReduce [5], an open source implementation of MapReduce [6], a distributed programming model to process large datasets; and HBase [7], a distributed, versioned, and column-oriented database. HDFS has been also used outside Hadoop to store results of scientific experiments [8] and as a general-purpose file system [9].

Before the version 2.0.0-alpha, HDFS had an undesirable characteristic that would cause an adverse impact on real-time 24x7 applications. HDFS belongs to the category of cluster-based parallel file systems. Such file systems are characterized by decoupling the namespace information from the data. A master node, the metadata server, services clients and manages the file system tree. Slaves, the object storage servers, are responsible for storing the data. Files are striped and replicated in parallel to several object storage servers for reasons of reliability and scalability. That architecture makes the implementation straightforward and helps to ensure good performance. However, it has a single point of failure: the metadata server, the interface with clients and the only node hosting the namespace.

Along with HDFS, in the same category of file systems one can also find Lustre [10], Apple Xsan2 [11], Panasas [12], Ceph [13], and Google File System [4]. In those systems, the crash of the metadata server compromises the availability of those systems. In response, they have rendered different techniques to deal with that problem, including journaling, namespace partitioning and shadow masters. Being a widely used distributed file system, HDFS has also received proposals on how the availability of its master node could be improved [14], [15], [16], [17].

In this paper, we propose a simple and efficient approach to provide high availability to HDFS 0.21 that takes advantage of open source efforts. The solution evolves the Backup Node, the checkpointer helper of the metadata server, into a hot standby. It achieves that firstly by augmenting the already replicated state of Backup Node. For that, we reused the method introduced by a previous high availability solution, AvatarNodes [16]. The method consists on making the storage servers to send messages that were destined to the metadata server to the standby as well. Secondly, it implements an automatic failover mechanism based on Apache ZooKeeper [18], the distributed coordination service provided by the Apache Hadoop Project. Due the reutilization of the Backup Node the solution could be accomplished with just 1373 lines of code.

In order to verify whether the solution is applicable to real systems, we put it under loads dominated by metadata and I/O operations. In both cases, the degradation of performance imposed by the state replication did not reduce the data throughput by more than 15% on average if compared to the HDFS 0.21. Moreover, the created hot standby could switch to active metadata server in less than 100 milliseconds.

1060-9857/12 $26.00 © 2012 IEEE
DOI 10.1109/SRDS.2012.33

The remaining of this paper is organized as follows: Section II gives an overview of HDFS; Section III describes the proposed solution; Section IV presents and discusses the tests and results; Section V shows how high availability is handled by distributed file systems and proposals made for the HDFS, and compares the approach taken by us to others; Section VI lists our conclusions; and finally Section VII outlines future work.

II. HDFS Overview

HDFS [19] is a file system written in Java with a traditional hierarchical namespace. Because HDFS was designed to support data-intensive applications, it favors throughput over latency. That is accomplished by the relaxation of some POSIX semantics and a simple coherence model. That model allows a file to have many readers but only a single writer controlled by a lease mechanism. The single writer can only append data to files. Like in disk file systems, files in the HDFS are composed by blocks. HDFS usually replicates blocks to three DataNodes to ensure the consistency and high availability of data.

This section describes the HDFS 0.21, which we used to develop our solution. Its cluster is composed of a single NameNode, a Backup Node, several DataNodes, and clients. The communication between nodes is done primarily trough a custom remote procedure call protocol. The architecture of HDFS 0.21 and its data flows can be seen in Fig. 1.

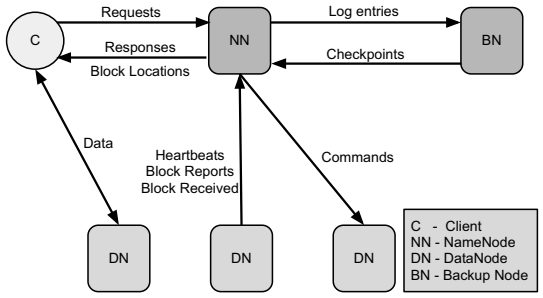


Fig. 1. HDFS 0.21: I/O flows and elements.

A. Client

Even though HDFS provides a POSIX-like API, applications must send their requests to the NameNode through a client. In order to prevent the NameNode from being a bottleneck, the client can directly contact the DataNodes for I/O operations. The client is responsible for calculating and verifying checksums whenever it reads or writes a block to detect data corruption.

B. DataNode

DataNodes are responsible for storing blocks. They accomplish that by recording the blocks as files of their local file system. DataNodes are constantly communicating their statuses to the NameNode trough messages. Those messages are:

- Heartbeat: this message is not only used to tell the NameNode that the node is still alive, but also to inform about its load and free space. The NameNode uses the response to heartbeats to send commands back to DataNodes. DataNodes whose heartbeats are not heard for a long time are marked as dead. By default, a DataNode sends a heartbeat at each three seconds.
- Block-Report: this message is a list of healthy blocks that the DataNode can offer. By default, a DataNode sends a block-report at each hour.
- Block-Received: on the occasion the DataNode receives a new block, sending a block-report would be cumbersome. Hence, in this case, the DataNode sends this shorter message to notify about recently received blocks.

Those last two messages enable the NameNode to learn the locations of all blocks within the HDFS cluster in addition to how many replicas of each block are available.

C. NameNode

The NameNode is responsible for:

- Managing the namespace: the file system tree and its associated metadata;
- Controlling access to files: HDFS supports POSIX-like permissions and requires leases for writers;
- Replica management: deciding where replicas are going to be placed, and identifying under and over-replicated blocks;
- Mapping blocks to DataNodes: the NameNode keeps track of the location of each block in the system.

The NameNode keeps its whole state in the main memory for performance reasons. The state is composed of namespace, leases, and block locations. In order to have some resilience, the NameNode implements journaling using two local files. The first file contains a serialized form of the namespace and serves as a checkpoint. The second file acts as a transactional log. Before updating the in-memory data structures, the NameNode records any changes on the namespace to the transactional log. The NameNode flushes and syncs the log before returning a response to a client. The NameNode does not persist the lease state and block locations because they are volatile data. Leases are valid only for a small period. DataNodes are reported to fail at considerable rates (2-3 failures per thousands nodes per day [3]), so replication status can change fast. Thus, any information about leases and block locations is lost in the event of a NameNode failure.

During the startup, the NameNode loads the last checkpoint to memory and replays the transactional log to create a new checkpoint. When that task is done, the node truncates the log and starts to listen to requests. However, at this moment, only read-only metadata requests are guaranteed to be successful, as the NameNode is in a state know as *safe mode*. The safe mode gives the NameNode enough time to be contacted by the DataNodes in order to recreate the block-to-DataNodes mapping. While in safe mode, writes and replication operations are not allowed. The NameNode is kept on safe mode, waiting

for block-reports, until a minimal replication condition is met. That condition is reached when a user-defined percentage of the total blocks in the system are reported to have the minimum allowed of replicas. After that, the NameNode can enter its regular operation mode.

D. Backup Node

As mentioned in the previous subsection, the NameNode only creates a checkpoint during its initialization. If no measure were taken, the transactional log would grow without bounds. That would mean that, in the next startup, the file system would be unavailable for a long time due to a prolonged processing of the log and generation of a new checkpoint.

To avoid that problem, HDFS has a checkpointer helper, the Backup Node. The Backup Node implements an efficient checkpoint strategy because it maintains its own view of the namespace synchronized with NameNode's. That avoids the necessity of regularly interrupting the NameNode to dump its complete state, as it would happen in traditional checkpointing techniques.

When the Backup Node starts, it registers itself to the NameNode. Then it obtains from NameNode its latest checkpoint and transactional log in order to create a new checkpoint. However, it does that only at that time and only with the purpose of synchronizing itself with the NameNode. Once it is synchronized, the NameNode sees the Backup Node as an additional storage area for the transactional log. So, when the NameNode writes a new entry to the log, that entry is propagated to the Backup Node. As soon as the Backup Node receives the entry, it applies the entry to its own in-memory data structures. Because of that and because the NameNode only responds to clients when all the log streams were flushed and synced, the Backup Node is ensured to keep an up-to-date view of the namespace. Consequently, checkpointing its own state is equivalent to checkpoint NameNode's. That is possible because both the Backup Node and the NameNode share most of their code: in terms of implementation, the Backup Node is a subclass of the NameNode. That also means that the Backup Node could potentially perform the NameNode's duties. Therefore, the Backup Node is a serious candidate to be part of a high availability solution for HDFS. In fact, it was designed with this possibility in mind [20]. We took advantage of that fact to develop the Hot Standby Node solution that is going to be described in the next section.

III. The Hot Standby Node Solution

A hot standby is a server node that at any given moment tries to maintain a complete and up-to-date copy of the state of its primary node. Consequently, when a failover system requests it to switch to an active role, it can do it in a short period of time, since there is little state to be recovered or reconstructed. In the previous section we mentioned that the Backup Node could be reused for a high availability solution. However, in order to turn it into a hot standby two steps need to be implemented: (i) extend the state already replicated by the Backup Node; (ii) build an automatic failover mechanism. Given that those two steps effectively accomplish our goal, we are going to call Hot Standby Node our modified version of the Backup Node. The next subsections are going to describe how we executed the steps, evaluating possible designs and justifying choices.

A. Extending State Replication

Recall the Backup Node keeps a synced view of the namespace with the NameNode. But, as described on Subsection II-C, the state of NameNode also comprises information about block locations and leases. Those two missing state components are required to process client requests. We adopted a different strategy for each component.

Block locations: For a HDFS cluster of 2,000 nodes and about 150 million files, the entire startup of NameNode takes about 45 minutes. A hypothetical high availability solution based on the Backup Node, with no additional state replication but capable of automatic failover, would still take 20 minutes to become able to serve clients [16]. That is the necessary time to process block-reports from all DataNodes in order to learn the locations of the blocks. From that example, it should be clear that the replication of the block locations is essential for a fast failover.

Two approaches can be taken here: (i) make the NameNode propagate any change on replicas to the Hot Standby Node or (ii) make the DataNodes also send their messages to the Hot Standby Node. The first approach has the benefit of keeping the knowledge about blocks synchronized between the NameNode and the Hot Standby Node. On the other hand, the data traffic between those nodes would be high. Messages can be consolidated and sent in batch to reduce the flow, but there are still other problems with this approach. It requires changes to a great extent of the NameNode's code. Furthermore, it imposes more CPU and memory pressure on the NameNode, harming its performance. In fact, the NameNode Cluster of China Mobile Research [21], described in Section V, employed this technique and they have observed 85% performance reduction for requests that only affect metadata.

The second approach has the benefit that the additional data traffic created by the duplication of the DataNode messages is approximately equally distributed among all the DataNodes, in opposition to the intense traffic between the NameNode and the Hot Standby Node of the previous approach. The drawback is that there is no guarantee that NameNode and Hot Standby Node will share a common view for the blocks in the system. However, some inconsistency is tolerable. If the Hot Standby Node misses some message, it may occasionally re-replicate a block, but data is never lost or corrupted.

We have chosen the second approach because it has a smaller impact on the NameNode and it shares the replication burden among the DataNodes. That was also the choice of AvatarNodes from Facebook [16], another high availability solution for HDFS. Since the well-tested code for their modification of the DataNodes was available [22], we decided to reuse it. An extra bonus from reusing that code is that it solves another issue. The NameNode only records to the transactional

log the blocks that constitute a recently created file when the file is closed. Consequently, the Hot Standby Node will not recognize any blocks created by in-progress write operations. Meanwhile, if it becomes active, the Hot Standby Node will consider those blocks invalid and it will request the DataNodes to delete them. To mend that situation, AvatarNodes added a list of blocks to the response for a block-received message. That list contains blocks that were present on the message but not recognized by its receiver. That allows to retry the block-received messages for those blocks until the file is closed.

As mentioned in the previous section, the NameNode uses replies to heartbeat messages to send commands to DataNodes. By extension, the same is true for the Hot Standby Node. In order to prevent it from sending commands that would conflict with the NameNode's orders, the Hot Standby Node is retained on safe mode until it becomes active.

Leases: Regarding the leases, we decided not to replicate the information about them. If the NameNode crashes while a file is being written, the new blocks added to that file will be lost. That will require the client to restart the write, generating a request for a new lease. Therefore, preserving the lease after the failover is not necessary since the old lease will not be reused. That behavior is tolerable for most applications using HDFS [23]. Hadoop MapReduce retries any tasks that failed. HBase only marks a transaction as completed when it issues a successful sync/flush call to persist the file contents. Furthermore, leases do not impact the time for the Hot Standby node to switch to active mode.

B. Implementing Automatic Failover

Once the NameNode fails, the Hot Standby must be instructed to assume the active role. Additionally, clients must be able to discover the new active server. For both cases we decide to use Apache ZooKeeper [18]. ZooKeeper is part of the Hadoop project, making it easily available to HDFS developers. ZooKeeper is a highly available distributed coordination service. It keeps a replicated database among a set of servers called ensemble. That database is structured as a tree whose nodes are called znodes. The znodes can store data. Any operation made over them is atomic. Users of ZooKeeper can set watchers on a znode to be informed of changes on it or in any of its children. A leaf znode can be created as ephemeral, i.e., it exists while the user session that created is still active, providing some liveness detection. The liveness detection is implemented through exchange of heartbeat messages between ZooKeeper servers and clients. Internal znodes can be used to create a group abstraction. All those features provide the basic constructs to implement several distributed protocols like membership management and leader election.

Failure Detection: The use of ZooKeeper as a failure detector involves one side for the ZooKeeper ensemble and other for the NameNode and the Hot Standby Node. On the side of the ZooKeeper ensemble, a znode is created with two purposes. The first purpose is to store the address of the current active server, initially the NameNode's network address. The second purpose is to serve as parent znode for the ephemeral nodes created by the NameNode and the Hot Standby Node. On the side of the NameNode and the Hot Standby Node, we introduced a new component, the Failover Manager. The Failover Manager takes care of maintaining a session with the ZooKeeper ensemble and creating the ephemeral znode that represents its server. Additionally for the Hot Standby Node, the Failover Manager sets a watcher on the ephemeral node of the NameNode.

When the NameNode crashes, the session it maintains with the ZooKeeper ensemble will eventually expire. This will cause its ephemeral node to be removed from Zookeeper's database, triggering the watcher set by the Hot Standby Node. At this moment, the Hot Standby Node will initiate the failover procedures that consist on:

1) Stopping the checkpointer thread. Like the Backup Node, the Hot Standby Node performs the duty of checkpointer helper until it becomes active;
2) Leaving the safe mode;
3) Updating the parent znode to point to its network address.

During failover clients reading or writing blocks will not notice the failure of the NameNode, because for those operations the client works directly with the DataNodes. The client will only receive a I/O exception when it has to contact the NameNode.

Active Server Discovery: HDFS clients must know the address of the active server. One way of transparently handling that is utilizing Virtual IP. Virtual IP is a technique that dynamically reassigns an IP address to another machine. So with Virtual IP, the IP address of the failed NameNode could be transferred to the Hot Standby Node. Clients would be able to continue issuing requests without noticing that, in fact, they are contacting the Hot Standby Node.

Unfortunately, Virtual IP is not available in all environments. Since we are using ZooKeeper for failure detection, we also decided to use it for active server discovery. So we have implemented the Active Server Discovery Library. As we mentioned above, the parent znode for the ephemeral znodes of the Namenode and the Hot Standby stores the address of the active metadata server and it is updated during a failover. The library only has to read and set a watcher on that znode to be capable of supplying up-to-date information.

C. Architecture

After all those design choices, the resulting architecture for the Hot Standby solution is depicted on Fig. 2.

The current implementation of Hot Standby Node solution tolerates only one fault, and it is not capable of reconfiguration. This choice simplified the design and eliminated the need for an election phase during the failover, thus making it faster. However, those limitations can be removed if ZooKeeper is used for leader election and the remaining Hot Standby Nodes are made to register with the new master.

We based our development on HDFS 0.21, the last stable release at the time we were developing the solution. The changes to the code of HDFS were small. Only 1107 lines of code

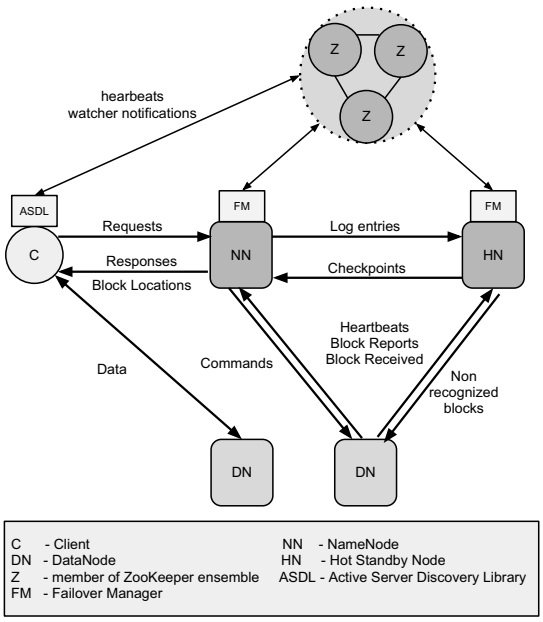


Fig. 2. Hot Standby Node Solution

were added, 45 modified, and 221 deleted. That represents only 0.18% of the original code. We used ZooKeeper 3.3.3 for the Failover Manager, Active Server Discover Library, and the ZooKeeper ensemble. The code for the solution as well the scripts and programs used on the tests are available at http://www.students.ic.unicamp.br/~ra078686/hotstandby.

IV. Experiments

In order to assess the performance overhead implied by our high availability solution, we compared it against the HDFS 0.21. Actually, we had to use a modified version which only creates checkpoints at startup. That was done in order to workaround a bug that would crash the Backup Node if the checkpoints were created during the execution [24]. In order to do the comparison, we used two different scenarios:

Metadata Scenario: load dominated by metadata operations. Each client creates and reads 200 files of one block each.

I/O Scenario: load dominated by I/O operations. Each client creates and reads a single file of 200 blocks.

The chosen environment to execute the tests was the Amazon Elastic Compute Cloud (EC2) [25]. That choice gave us the flexibility to test with several machines, making the tests closer to a real usage of HDFS. Besides that, it makes easier for other researchers to reproduce the experiments and verify the results. The unpredictable load of Amazon EC2 could affect experiments, leading to variation on results. However, the tests showed them to be small. We used 42 virtual machine instances. Twenty of those were used to run the DataNodes and another twenty were used to run one client each. One instance ran the NameNode process. The remaining instance would either run the Backup Node when testing the HDFS 0.21 or the Hot Standby Node when testing our solution. We used one extra instance to run a ZooKeeper server in the tests with the high availability solution. All the virtual instances were of type small [26] and ran Ubuntu 10.04 LTS 32 bits with Java 6. The block size used was the default, i.e., 64MB. Since the tests used 200 blocks, it was possible to fill one quarter of the total storage the cluster.

With the described set up, the test procedure was:

1) Start the ZooKeeper server if testing the Hot Standby Node solution;
2) Start the NameNode;
3) Start the Backup Node if testing the HDFS 0.21 or the Hot Standby Node if testing the solution;
4) Start the DataNodes;
5) Start the clients;
6) Wait clients to finish their jobs;
7) If testing the Hot Standby Node solution, shut down the NameNode and wait until the failover is complete.

For each permutation we took five samples, yielding a total of 20 runs (2 HDFS configurations x 2 scenarios x 5 samples). In order to be possible to analyze the results, we assumed that the clocks of all virtual machine instances were partially synchronized, and that any clock skews were negligible. It is a fair assumption considering that servers at Amazon EC2 run NTP to update their clocks. Based on that assumption, we defined the period of time to be used on the analysis as the one starting with the first client joining the cluster and ending with the last client leaving. In other words, it was the period that the system was under load. Table I displays the average period of analysis along with its standard deviation for all permutations.

TABLE I
Average period of analysis in seconds for the experiments.

Scenario	Configuration	Period
Metadata	HDFS 0.21	$(3,788 \pm 398)s$
	Hot Standby	$(3,963 \pm 180)s$
I/O	HDFS 0.21	$(4,192 \pm 242)s$
	Hot Standby	$(4,027 \pm 270)s$

The follow subsections presents results that are essential to comprehend the impact of the Hot Standby Node on the system. For all plots, the error bars represent the standard deviation for the samples. Solid bars ( ) refer to the HDFS 0.21, whereas patterned bars ( ) refer to our solution. The results for the metadata scenario are represented by white bars ( ), while the I/O results are represented by gray bars ( ). A bar labelled "Backup Node or Hot Stanby Node" means that it refers to the Backup Node when the HDFS 0.21 is under test (solid bar) or to the Hot Standby Node when our solution is being tested (patterned bar).

A. Network Flow for the DataNodes

Fig. 3 displays the average data flow that a DataNode sent and received during the period of analysis. The values were

measured on the Ethernet interface of the instances running the DataNodes. We assumed that traffic generated by the other processes were negligible.

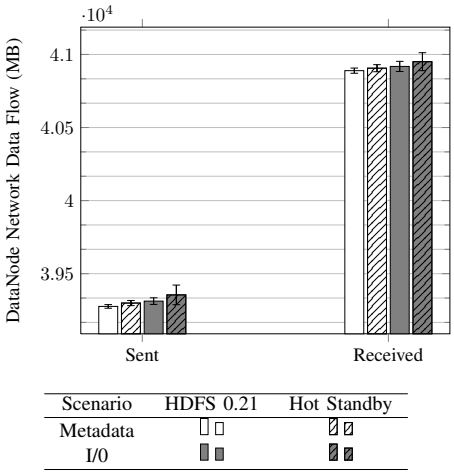


Fig. 3. Network data flow for DataNodes.

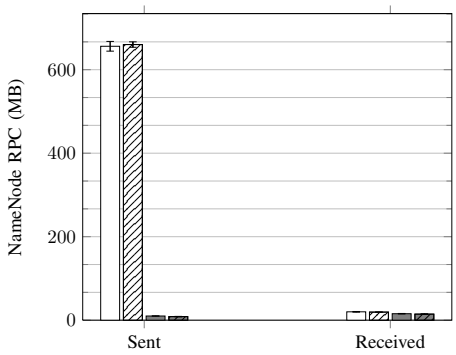
The data flow is approximately the same in all situations. Intuitively, one would expect an increase in the outgoing flow for our solution due the extra messages sent to the Hot Standby Node. Nevertheless, those messages account for less than 0.43% of total traffic out of a DataNode. The messages were diluted in the flow generated by clients reading the blocks.

B. RPC Data Flow for the Master Server Nodes

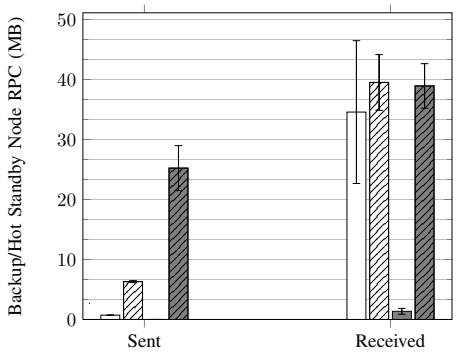
The data flows for the NameNode and the Backup Node (and its replacement in our solution, the Hot Standby Node) are much lower compared to DataNodes' flows. They are almost exclusively generated by remote procedure calls (RPC). Thus, we focus the attention on the RPC traffic. Fig. 4 shows the amount of data sent and received through RPC by the master server nodes (NameNode, Backup Node and Hot Standby Node) during the period of analysis.

Fig. 4(a) does not reveal any considerable difference between the NameNode from HDFS 0.21 and the one from our solution. This is not surprising, as NameNode's code has not changed. The high volume of data sent in the metadata scenario is mostly due the log entries being sent to the Backup Node or the Hot Standby Node. We can verify that on Fig. 4(b), in which, in the same scenario, high amounts of data are received by the Backup Node and the Hot Standby Node. Still in that figure, there is an increase of RPC traffic if we compare the Backup Node (solid bars) to the Hot Standby Node (patterned bars), because the latter interacts with the DataNodes besides processing the transactional log.

In the I/0 scenario, the volume of data sent by the Hot Standby Node is bigger than in the metadata scenario, even though the node has to exchange messages with the DataNodes for the same number of blocks. In the same scenario, the amount of data received by the Hot Standby Node achieves the same levels than the ones of the metadata scenario, despite



(a) RPC data flow for the NameNode



(b) RPC data flow for the Backup/Hot Standby Node

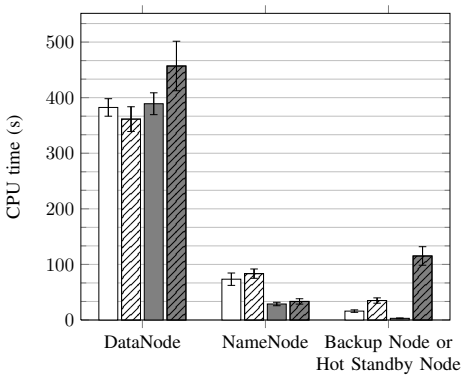
Scenario	HDFS 0.21	Hot Standby
Metadata		
I/0		

Fig. 4. RPC data flow for the master server nodes.

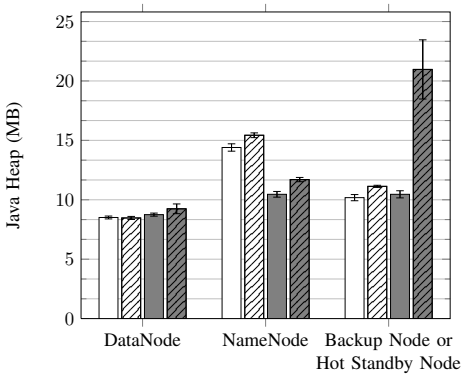
the fact that the incoming traffic from NameNode is smaller in that case. The reason behind those unexpected results lies on the block-received messages. Recall that the Hot Standby Node returns to the DataNode a list of blocks present on the block-received message that were not recognized by it. On its turn, the DataNode enqueues those blocks to be sent again in the next block-received message. In the metadata scenario, the files are one block long. So, when the block-received message reaches the Hot Standby Node, the file is very likely to be already closed. Thus, the Hot Standby Node recognizes almost all the blocks in the first attempt. The block-received messages are then kept short with an average size of 465 bytes. So the Hot Standby Node's responses to them, with an average size of 165 bytes. The same does not happen on the I/O oriented scenario. The files are composed by 200 blocks. Therefore, only after the 200th block is written is that the file is closed and that any of its blocks present on a block-received message will be recognized. So, as the files are written but not closed, the block-received messages keep growing in size, so do the responses. The average size for the block-received message jumps to 7,830 bytes and the response to 5,703 bytes.

C. CPU time and Memory Consumption

Fig. 5(a) shows the amount in seconds of CPU time spent to run a HDFS server process during the period of analysis. In the case of DataNodes, the result refers to the average CPU time among them. Fig. 5(b) reveals Java Heap usage during the same period. The results come from the average of samples taken at each ten seconds. The bars for the DataNode reflect the average value among the averages of each DataNode in the system.



(a) CPU usage for server nodes



(b) Java Heap usage for server nodes

Scenario	HDFS 0.21	Hot Standby
Metadata		
I/0		

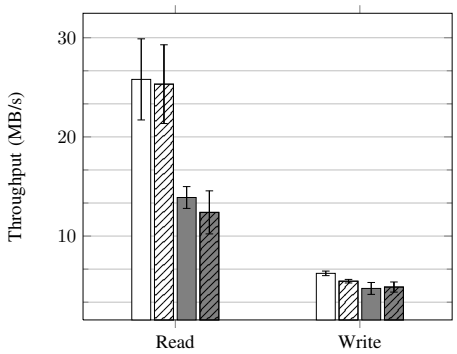
Fig. 5. CPU time and heap memory consumption for nodes

Turning the attention to the NameNode, there is a small increase in CPU and memory utilization because our high availability solution introduces a new component, the Failover Manager, which is constantly communicating with the ZooKeeper server.

The DataNodes present similar use of CPU and Heap Memory for all configuration and scenarios, except by in the I/O scenario, in which there is some growth in the resource consumption. That is also a consequence of block-received message problem mentioned earlier. However, the node most impacted by that problem is certainly the Hot Standby Node. That fact is denounced by its pronounced bars in both plots. The longer messages require more processing time, and while they are processed, a considerable number of objects representing the blocks are created and discarded very quickly. That not only increases the use of the heap, but also forces the garbage collection algorithm to run many more times. On average, there were 260 runs of the garbage collector for the metadata scenario against 1,998 runs for the I/O scenario. The more frequent execution of the garbage collector generated even more CPU pressure.

D. Read and Write Throughputs

We consider the read and write throughputs to be the two most relevant metrics to evaluate the impact of our solution. That is because they dictate the amount of work that can be performed on behalf of the clients. The throughput was measured diving the file size by the time spent by the client to read or write it. The bars of Fig. 6 show the average read and write throughputs for all scenarios and configurations.



Scenario	HDFS 0.21	Hot Standby
Metadata		
I/0		

Fig. 6. Data throughput at clients

In fact, Fig. 6 shows reductions for both read and write throughputs in comparison to the HDFS 0.21 as it would be expected. However, the reduction is always smaller than 2 MB/s.

E. Transition Time

Other important measure is the failover time. The failover time can be said as time for the failure detection plus the time for switching the Hot Standby Node to the active mode. The time for failure detection is very dependent on environment factors such as the network latency and virtualization. As greater is the impact of these factors, the greater will be the necessary timeout to prevent an incorrect detection of failure. In fact, while testing on our LAN, the ZooKeeper session timeout could be safely set to 500ms. However, for the tests at Amazon EC2 it had to be set for 2 minutes in order to avoid false positives. Thus, we focused the measurements on the time

for switching the Hot Standby Node, since it is the component of failover time that can be impacted by our implementation. The average time to switch the Hot Standby Node to active mode on both test scenarios is showed on Table III.

TABLE III
THE TIME TO SWITCH THE HOT STANDBY NODE TO THE ACTIVE MODE.

Scenario	Switch time	Standard Deviation
Metadata	56.60ms	35.61ms
I/O	57.60ms	18.20ms

The time to switch the Hot Standby Node to active mode was about the same for both scenarios, as the replication of the namespace and the block locations are not dependent on the characteristics of the workload.

V. RELATED WORK ON HIGH AVAILABILITY

This section describes high availability strategies adopted by different distributed file systems, proposed strategies to HDFS, and the solution recently incorporated into HDFS.

A. Cluster-based File Systems

Lustre [10] uses journaling. Every change to the file system's metadata is recorded to a transactional log. So, in case of failure of the metadata server, the log can be replayed, allowing it to return to the last consistent state. Besides that, Lustre uses an active-passive pair strategy. Lustre has an active metadata server and a standby one sharing the transactional log storage. The standby metadata server is constantly monitoring the active server, so when it notices the failure of the latter, the standby server can take over using the transactional log from the shared storage. That approach is taken further by Apple Xsan2, which supports multiple standbys. Priorities can be assigned to each of the standbys, so the standby with the higher priority is the one that will assume the active role.

In addition to the techniques implemented by Lustre, Panasas [12] does namespace partitioning: the namespace is split among several metadata servers, each one responsible for a disjoint sub-tree of the file system. Although the primary goal for this technique is to improve the scalability of the system, it can be thought that it creates compartmentalized fault domains. Only the sub-tree managed by the failed metadata server will become unavailable.

Ceph [13] extends namespace partitioning by making it dynamic. Depending on the current load, entire sub-trees of the file system can be reallocated or even replicated to other metadata servers. That keeps the load of the file system balanced and promotes high availability.

Google File System [4] offers shadow masters, read-only metadata servers. They are called "shadow" because they update themselves by reading the primary metadata server's log and thus they may lag the state of the latter. However, that does not prevent them from replying read-only requests even if the primary is down.

Table II compares the characteristics of the high availability solutions employed by the mentioned distributed file systems, including HDFS.

B. Proposals for HDFS

The simplest approach to improve the availability of the NameNode, not requiring code changes, was developed by ContextWeb [14], a web advertising company. That solution employs an active-passive pair strategy and Distributed Replicated Block Device (DRBD) [15] to store the transactional log. DRDB is a Linux logical device block implemented in such way that any data written to it is mirrored over a TCP connection to another storage device, thus yielding transparent data replication for applications. The monitoring of the NameNode is done by Linux-HA [31], a cluster resource management tool. When Linux-HA detects the failure of the NameNode process, it starts another one in the standby's machine. Thanks to DRDB, the standby's copy of the transactional log is kept up-to-date, enabling the newly created process to recover the file system to its last consistent state.

Facebook has also developed an active-passive pair solution called Avatar Nodes [16]. In Avatar Nodes, the standby NameNode is a live process that keeps reading the transactional log from the shared storage in order to maintain its namespace view synchronized with the active NameNode. As mentioned

TABLE II
HIGH-AVAILABILITY TECHNIQUES USED BY DIFFERENT CLUSTER-BASED DISTRIBUTED FILE SYSTEMS.

File Sytem	Journaling	Active-Passive Pair	Multiple Standbys	Namespace Partitioning	Read-Only Replicas
Apple Xsan2	Yes	Yes	Yes	Static	
Ceph	Yes			Dynamic	
Google File System	Yes	Yes		Static	Yes
HDFS 0.21	Yes				
HDFS 2.0.0	Yes	Yes		Static	
Lustre	Yes	Yes			
Panasas	Yes	Yes		Static	

in the previous sections, this solution modified the DataNodes to also send messages to the standby NameNode. That enables the standby NameNode to have information about block replicas, ensuring a fast failover. They also managed to log blocks allocation as they are made, so in progress writes will not be lost. Nevertheless, AvatarNodes lack automatic failover, as it is targeted for preventive maintenance and software upgrades.

Both IBM China and China Mobile Research bet on a more traditional passive replication strategy of the NameNode by employing state-update messages [32]. In IBM China's solution [17], the messages not only carry the namespace changes but also the lease information. They argue that replicating more state information would considerably impact performance. The failover mechanism is built-in and based on heartbeat messages. In contrast, the NameNode Cluster of China Mobile Research [21] opts to replicate all the possible state information. The overhead is reduced by making the active NameNode consolidate as much as possible the messages from DataNodes. That allows the number of state-update messages to be reduced. NameNode Cluster uses Linux-HA for automatic failover. Both solutions support multiple standbys.

Clement et al. have developed the UpRight library [29] to add protection against Byzantine failures to systems that are already crash fault tolerant. In order to prove that they have achieved their goal, they changed HDFS to use their library. In their solution the UpRight library effectively replaces the entire communication layer of the system. All messages are now ordered and routed by the UpRight cluster. Other step they took was removing all the non-determinisms from the execution path of the NameNode. With globally ordered messages being sent to a set of deterministic NameNodes, they could accomplish a high availability solution based on state-machine replication [33].

C. Official Solution

The support for high availability introduced in the HDFS 2.0.0-alpha on May 2012 [30] follows the same design lines of AvatarNodes, with shared storage for the log and duplicated messages. There is a fencing mechanism to preventing the two nodes from writing to the log in the shared storage at the same time. HDFS 2.0.0-alpha has only manual failover, but that limitation has been solved on the development branch [34]. That version also allows several NameNodes, each one responsible for a different file system, to share the same pool of DataNodes. Clients are capable to see all the different file systems as single virtual namespace.

Table IV compares the different proposals including the one made by this paper. Compared to them, our solution do not use components that are external to the Hadoop Project or bring significant changes to the HDFS's code. That choice results in simpler configuration and smaller maintenance effort. Our NameNode does not need to be deterministic, so it can take advantage of parallelism in its full potential. We do not employ shared storage or similar as we understand that it would transfer the high availability problem to the shared storage. Although our solution has automatic failover, failures on AvatarNodes and HDFS 2.0.0-alpha are more transparent to applications due changes on the client and logging mechanism. Like them, our solution tolerates only one fault, contrasting with solutions from IBM China and China Mobile Research that are more fault-tolerant.

VI. Conclusions

The high availability of the Hadoop Distributed File system is a key interesting problem as the Hadoop software is being more and more employed on real-time systems. In that sense, this paper introduced the Hot Standby Node solution. Despite the overhead created by the state replication, the solution could still deliver a good throughput to the clients no mattering

TABLE IV
Comparative table of proposals of high availability solutions for HDFS.

Proposal	Replication strategy	Replicated state	Failover	Changed lines
ContextWeb	DRBD	only namespace	Linux-HA	zero
AvatarNodes	shared storage and duplicated DataNode messages	namespace, lease and block information	manual	8120 [27]
IBM China	update-state messages	namespace and leases	built-in	unknown
Namenode Cluster	update-state messages	namespace, lease and block information	Linux-HA	5570 [28]
UpRight HDFS	state-machine replication	namespace, lease and block information	not applicable	2696 [29]
HDFS 2.0.0-alpha	shared storage and duplicated DataNode messages	namespace, lease and block information	manual	15,000+ [30]
Hot Standby Node	log streaming and duplicated DataNode messages	namespace and block information	ZooKeeper	1373

the kind of the load. In any case, the Hot Standby Node could assume the NameNode's position in a small period of time. The replication strategy adopted by the presented solution has the advantage of sharing the replication burden among the other elements of HDFS, instead of concentrating it on the NameNode. This lessens the impact on performance. By reusing the Backup Node, we kept the high availability problem in the HDFS domain. That also enabled the changes to be small, easy to maintain and be understood by other developers and researchers. Finally, because we are only using components from the Hadoop project, the Hot Standby Node solution is still easy to deploy, as those components are very well integrated.

VII. Future Work

The experimental results have showed that there is still room for improvements. The analysis of Section IV showed the need to reduce the size of the block-received messages in order to minimize resource consumption. To accomplish that the Hot Standby Node should be aware of which blocks constitute a file as early as possible. That will also address other limitation of the current approach. Since the Hot Standby node will know the blocks of a file before it is closed, it will be possible to continue an in-progress write after the NameNode fails. That will require the replication of leases and changes to the HDFS client. Finally, we also intend to support multiple standbys and reconfiguration.

Acknowledgments

André Oriani would like to thank his employer, Motorola Mobility, for allowing him to use four of his week working hours on this research. The authors would like to thank Rodrigo Schmidt from Facebook for suggesting this research topic and providing good insights during its development.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 3-7 2010, pp. 1 –10.

[2] "Facebook has the world's largest Hadoop cluster!" http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html, last access on July 10th, 2012.

[3] K. Shvachko, "HDFS Scability: The Limits to Growth," *;login: The Usenix Magazine*, vol. 35, no. 2, pp. 6–16, April 2010.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[5] "Apache Hadoop," http://hadoop.apache.org/, last access on July 10th, 2012.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004, pp. 137–150.

[7] "HBase," http://hbase.apache.org/, last access on July 10th, 2012.

[8] B. Bockelman, "Using Hadoop as a grid storage element," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012047, 2009. [Online]. Available: http://stacks.iop.org/1742-6596/180/i=1/a=012047

[9] D. Borthakur, "HDFS High Availability," http://hadoopblog.blogspot.com/2009/11/hdfs-high-availability.html, last access on July 10th, 2012.

[10] "Lustre," http://www.lustre.org, last access on July 10th, 2012.

[11] "Apple Xsan2," http://images.apple.com/xsan/docs/L363053A_Xsan2_TO.pdf, last access on July 10th, 2012.

[12] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–17.

[13] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a Scalable, High-performance Distributed File System," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320.

[14] C. Bisciglia, "Hadoop HA Configuration," http://www.cloudera.com/blog/2009/07/hadoop-ha-configuration/, last access on July 10th, 2012.

[15] "DRBD," http://www.drbd.org, last access on July 10th, 2012.

[16] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache Hadoop Goes Realtime at Facebook," in *Proceedings of the 2011 international conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1071–1080. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989438

[17] F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li, "Hadoop High Availability through Metadata Replication," in *CloudDB '09: Proceeding of the first international workshop on Cloud data management*. New York, NY, USA: ACM, 2009, pp. 37–44.

[18] "Apache Zookeeper," http://hadoop.apache.org/zookeeper/, last access on July 10th, 2012.

[19] "HDFS architecture," http://hadoop.apache.org/common/docs/current/hdfs_design.html, last access on July 10th, 2012.

[20] "Streaming Edits to a Standby Name-Node," http://issues.apache.org/jira/browse/HADOOP-4539, last access on July 10th, 2012.

[21] X. Wang, "Pratice of NameNode Cluster for HDFS HA," http://gnawux.info/hadoop/2010/01/pratice-of-namenode-cluster-for-hdfs-ha/, last access on July 10th, 2012.

[22] "Hot Standby for NameNode," http://issues.apache.org/jira/browse/HDFS-976, last access on July 10th, 2012.

[23] D. Borthakur, "Hadoop AvatarNode High Availability," http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html, last access on July 10th, 2012.

[24] "When checkpointing by backup node occurs parallely when a file is being closed by a client then Exception occurs saying no journal streams." http://issues.apache.org/jira/browse/HDFS-1989, last access on July 10th, 2012.

[25] "Amazon EC2," http://aws.amazon.com/ec2/, last access on July 10th, 2012.

[26] "Amazon EC2 Instance Types," http://aws.amazon.com/ec2/instance-types/, last access on July 10th, 2012.

[27] "AvatarNodes implementation at Facebook's code repository at Github," https://github.com/facebook/hadoop-20/tree/master/src/contrib/highavailability, last access on July 10th, 2012.

[28] "NameNode cluster's code repository at Github," https://github.com/gnawux/hadoop-cmri/, last access on July 10th, 2012.

[29] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "UpRight Cluster Services," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 277–290.

[30] "High Availability Framework for HDFS NN," https://issues.apache.org/jira/browse/HDFS-1623, last access on July 10th, 2012.

[31] "Linux-HA," http://www.linux-ha.org/, last access on July 10th, 2012.

[32] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, *The Primary-Backup Approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 199–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=302430.302438

[33] F. B. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: a Tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[34] "Automatic failover support for NN HA," http://issues.apache.org/jira/browse/HDFS-3042, last access on July 10th, 2012.