INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

The Search for a Highly-Available Hadoop Distributed Filesystem

André Oriani *Islene C. Garcia*
Rodrigo Schmidt

Technical Report - IC-10-24 - Relatório Técnico

August - 2010 - Agosto

The Search for a Highly-Available Hadoop Distributed Filesystem

André Oriani[*1], Islene Calciolari Garcia[†1], and Rodrigo Schmidt[‡2]

[1]Institute of Computing, State University of Campinas, Campinas, Sao Paulo, Brazil
[2]Facebook, Inc., Palo Alto, CA, USA

Abstract

Hadoop is becoming the standard framework to process large amounts of data. It takes advantage of distributed computing to do it fast and efficiently. However, this is only possible if data is supplied with high availability and consistency. Those goals are fulfilled by a core piece of Hadoop called the Hadoop Distributed Filesystem (HDFS). HDFS is a very scalable distributed filesystem capable of storing petabytes and providing high throughput data access. It makes intensive use of replication and checksums to protect the system from data loss and corruption. Despite of all those qualities, HDFS has a central component whose maintenance demands the entire system to be shut down. Furthermore, that component is also a single point of failure. Those limitations make HDFS unsuitable for 24x7 applications. In this technical report we are going to make a high-level introduction to HDFS and discuss attempts to solve the mentioned problem.

[*]`aoriani@comp.ufscar.br`
[†]`islene@ic.unicamp.br`
[‡]`rschmidt@facebook.com`

Contents

1 Introduction

From the content uploaded everyday to the Internet to the thousands of records generated by a single scientific experiment, data has never been produced in such large scale and so fast. To cite figures, in less than 4 years the number of pages indexed by Google jumped from 8 billions to 1 trillion. Everyday Google process 20PB of data. Every minute 20 hours of video are uploaded to Youtube and 270,000 words are typed in Blogger [36]. Facebook monthly receives 3 billions photos [34]. The Large Hadron Collider expects to produce 15 petabytes annually [52]. To finish this collection of statistics, the sequencing of a single human DNA generates roughly 90GB of raw data [43].

But this extremely huge amount of data is useless if it cannot be processed and analyzed to be turned really fast into information. The most cost-effective and scalable solution for such complex problem is distributed computing. And one of the most known distributed computing frameworks is Hadoop MapReduce. Hadoop MapReduce is one of the sub-projects of the Apache Hadoop Project that "develops open-source software for reliable, scalable, distributed computing" [29]. The project is mainly maintained by software engineers from companies like Facebook, Yahoo!, and Cloudera; although it receives many contributions from the open source, free software, and academic communities. The Hadoop software is used by several companies including Amazon, Adobe, AOL, Google, Hulu, IBM, Last.fm, LinkedIn, and Twitter. For instance, Amazon offers Hadoop MapReduce as part of their Elastic Cloud Computing (EC2) services.

Turning our attention back to Hadoop MapReduce. Hadoop MapReduce is the implementation of the MapReduce paradigm for distributed computing introduced by Google's employees in the seminal paper "MapReduce: simplified data processing on large clusters" [14]. MapReduce tackles the problem of processing a large input by dividing it into smaller independent units to be analyzed by system elements called mappers. The intermediate results are then combined by a set of reducers to get to a final answer to the computation. To have an idea of the processing power of MapReduce is suffice to say that Hadoop's implementation is the current winner of the Jim Gray's sorting benchmark, sorting one terabyte in just 62 seconds. Another great feat of Hadoop MapReduce was to convert four terabytes of scanned editions of The New York Times paper to PDF in less than 24 hours [28].

Of course, achievements like the mentioned ones require very large clusters. The terabyte sorting used one of the largest Hadoop clusters ever deployed. Yahoo! put together to work about 3800 nodes. Each node was a machine with two Quad Core Xeon processors, 16GB of RAM memory, and 4 SATA disks of 1 TB each. At the moment, Facebook claims to have the largest Hadoop cluster in terms of storage [35]. The cluster has a storage capacity of 21PB and it is composed by 1200 eight-core machines plus 800 sixteen-core machines, each one with 32 GB of RAM and at least 12TB for the combined capacity of hard disks.

For Hadoop MapReduce to deliver its performance it is crucial to have a storage system that can scale in same degree as it and still provide very efficient data retrieval with high throughput. That storage system is the Hadoop Distributed Filesystem(HDFS) [28, 19]. HDFS has proved to be able to store more than 14PB, to offer read and write throughputs higher than 40 MB/s and to handle 15,000 clients [22]. But Hadoop MapReduce is not the only one to put its stakes on HDFS. HBase [38, 28], another subproject of Hadoop, is a distributed, versioned and column-oriented database built on top of HDFS after Google's BigTable [11]. HBase is able to support tables with billions of rows and million of columns. HBase extends HDFS by supporting small records and random read/write access.

Nowadays, HDFS is becoming even more popular, being used not only to support Hadoop MapReduce, HBase or any HPC tasks but also to serve a broader range of applications varying from multimedia servers to sensor networks [7]. Even the scientific community has acknowledged its value to store huge result sets from experiments [3]. The increasing popularity as general-purpose storage system for large scale is due its reliability, consistence, and high availability of data. Those qualities are the product of the intensive use of replication and checksums by HDFS.

By contrast, the overall high availability of HDFS is somewhat questionable. As we going to see later, HDFS belongs to the class of distributed filesystems that decouples the namespace information from data. That means that in HDFS one node manages the filesystem tree while the remaining nodes store the data. That design choice considerably simplifies the implementation but it has the downside of creating a single point of failure for the system. Consequently, large maintenances or upgrades require the entire system to be taken down. In the worst-case scenario, a single failure can bring the filesystem into a non-operating state or make it completely lost. And those situations do happen. ContextWeb had 6 events of that kind in the period dating from April 2008 to October 2009, and only half of them were planned [16]. As HDFS is beginning to be employed by online and mission-critical applications, periods of unavailability of the filesystem are becoming less tolerable.

Thus, there is an increasing interest over the matter and Hadoop users and developers are proposing some solutions to the problem. In this technical report, we intend to expose and analyze them. For that, the remaining sections of this paper are organized as follows. In section 2 we describe the architecture and implementation of HDFS. Section 3 characterizes the runtime state of HDFS, categorizes high avaiability solutions, and finishes outlining and analysing proposals of solution. Section 4 summarizes how other distributed system deal with the high availability problem. Finally, in section 5, we conclude the paper and talk about future works.

2 An Overview of HDFS

HDFS arose in 2004, when Doug Cutting and Mike Cafarella were working in the Nutch project [10], a web search engine. Nutch was generating fast growing web crawling indexes and managing all that data was getting more and more difficult. Fortunately for them, googlers had recently published a paper entitled "The Google File System" [17], which described the storage system used by Google to store the huge amount of data generated everyday at their datacenters. Realizing that the new distributed filesystem would solve their problem, Cutting and Cafarella start working in an open source implementation of GFS called the Nutch Distributed Filesystem(NDFS). Later, NDFS was incorporated to the Apache Hadoop Project and renamed to its current name.

HDFS is designed to a very scalable filesystem running in non-expensive commodity machines. Practical limits today account for 21PB, 4000 nodes, 15000 clients and 60 million files [22, 35]. That great number of parts involved creates an environment with a non-trivial probability for failures. Thus, HDFS also endorses GFS' motto "*failures are the norm rather than exception*" [17] by making intensive use of replication and checksums to protect data from loss and corruption.

Another fact that influenced the design of HDFS was that Hadoop MapReduce jobs require large streaming reads. So the filesystem favors high data throughput over latency. That is accomplished with the relaxation of some POSIX semantics [49] and a simple coherency model that allows many simultaneous readers but only one writer at time.

2.1 Architecture

The architecture of HDFS, depicted on Figure 1, is very straightforward. HDFS has a master-slave architecture with a single master and several slaves. Files are split in equal-sized chunks, the blocks, what simplifies the storage management. The master node, called namenode, manages the namespace and metadata. The slave nodes, the datanodes, are responsible for storing the blocks. The HDFS client offers a POSIX-like interface. Even though all operations are requested to the namenode, all data flows occur directly among datananodes and clients, so the namenode does not become a bottleneck.

We then start detailing the system entities:

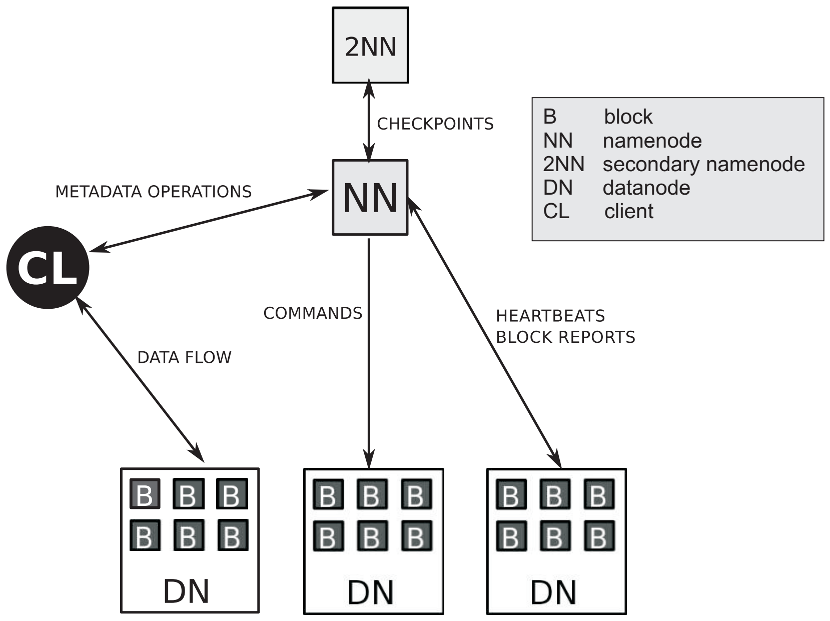

Figure 1: The HDFS flows and its elements.

Blocks Disk file systems are based on the abstraction that files are composed by blocks, the smallest data unit for transfers. The same applies to files on HDFS. But unlike their disks counterparts, which are a few kilobytes in size, blocks on HDFS are much larger. The default block size is 64 MB, although 128 MB is becoming quite common. The rationale behind that choice is to minimize the cost of seeks. As we have already said, HDFS is optimized for streaming data access, so the time to read the whole file is more important than the latency for reading the first bytes. By making the block big enough such that its transfer from disk takes considerable longer than seeking for its start, the file transfer can occur at disk transfer rate.

The block abstraction also brings other benefits. Files can be arbitrarily large, bigger than any single disk in the cluster, because blocks can be scattered among several datanodes. Provided that they have fixed size, storage management is simplified a lot because it is trivial to know how many blocks fit in a datanode. Blocks on HDFS are replicated. So if one gets corrupted or missing, data can still be read from another replica. Thus block replication delivers high fault tolerance and data availability. Replication also alleviates the read load because highly used files can have higher replication factor, which enables reads from the same file to be made in parallel using different copies.

Blocks are represented on datanodes by two local files. One holds the raw data and the other holds the block metadata. The block metadata is comprised of version, type information and a series of checksums for sections of the block. No information regarding the file owning the block is kept on datanodes. Still regarding the block size, one may argue that such large size would lead to a serious internal fragmentation problem. However, the file storing the block's data is so big as the block's content. The data is appended to it until the block size is reached, when then a new block is allocated.

Datanodes The datanodes are responsible for storing blocks. Periodically the datanode will send block reports and heartbeat messages to the namenode. A block report is a list of healthy blocks that the datanode can offer. The sending of block reports is randomized in order to prevent the namenode from being flooded with those reports. Heartbeat messages, in its turn, tell the namenode the node is still alive. The acknowledgment for those messages is used by the namenode to issue commands on datanodes. Examples of commands may be to replicate a block to other nodes, to remove local replicas, to re-register or to shut down the node, or even to send an urgent block report. Nodes whose heartbeats are not heard for a long time are marked as dead. No request will be redirected to them and the blocks they hold are assumed to be lost. Occasionally, the datanode runs a block scanner to verify all blocks for data corruption.

Namenode The namenode is the main node in the system, being responsible for:

- Managing the filesystem tree and metadata including name, size, permissions, ownership, modification and access times, as well as the mapping from files to blocks. HDFS has a traditional hierarchical namespace composed by files and directories. In contrast to most popular UNIX-based filesystems, in which directories are special types of files, in HDFS they are implemented as just some piece of filesystem metadata, stored at the namenode only.

- Controlling access to files by allowing a single writer and many readers. POSIX-like permissions and some sort of storage quotas are supported. To simplify its design and implementation only append writes are permitted.

- Replica management, deciding where replicas are going to be placed, identifying under-replicated blocks, and coordinating replication with datanodes.

- Mapping blocks to datanodes. The namenode keeps track of the location of each block replica in the system. That information is not persisted on disk but recreated every time the namenode starts up.

The namenode keeps an in-memory representation of the whole filesystem structure and metadata. Consequently, the number of files is limited by the amount of RAM memory available in the machine running the namenode process. In order to have some resilience, the namenode makes use of two local files. The first file, *fsimage*, contains a serialized form of the filesystem tree and metadata, and serves as checkpoint. The second one, the *edits*, acts like a transaction log. Prior to updating the in-memory data structures, the namenode records the operation to the *edits* log, which is flushed and synced before a result is returned to client.

When it starts, the first thing namenode does is to load the last checkpoint to memory. Then it applies all operations from the *edits* log, creating a new checkpoint. When that task is done, the log is truncated and the node starts to listen to RPC and HTTP requests. However, at this moment only read requests to the filesystem metadata are guaranteed to be successful, as the namenode is in a state know as the safe mode. The safe mode gives the namenode enough time to be contacted by datanodes in order to recreate the block-to-datanodes mapping. While in safe mode, write and replication operations cannot occur. The namenode waits for block reports from datanodes until a minimal replication condition is attended. That condition is reached when a user-defined percentage of the total blocks in the system are reported to have available the minimum allowed of replicas. After that, the namenode still waits for a while, when then it enters in the regular-operation mode.

Since the namenode is the only node aware of the namespace, in the event of a catastrophic failure of the machine hosting it, the entire filesystem is lost. It is easy to see now that the namenode is a single point failure for the system. As countermeasure for such a bad failure scenario, the namenode's persistent state is usually stored in multiple disks, including NFS shares.

Secondary Namenode Recall that the namenode only creates a checkpoint during the initialization. If nothing were done, the transactional log could grow without limit. As result, the creation of a new checkpoint at start up would let the system unavailable for a very long time.

The secondary namenode solves the matter by acting as a checkpointer. At regular intervals, the secondary namenode will ask the namenode to roll the transaction log and start a new one. Then it retrieves the current checkpoint and the rolled log from the namenode. The checkpoint is loaded to the secondary namenode's main memory and the transactions from the log are applied to it in order to yield a new checkpoint. The newly-created checkpoint is sent back to the namenode. So, at the end of this process, the namenode has an up-to-date *fsimage* and a shorter *edits*.

From the procedure above, it is clear that the memory requirements for the secondary namenode are very similar to the namenode's one. So it is advisable to run it on a separate machine. The use of the secondary namenode avoids the fine grain lock of the namenode's data structures or the need to implement a copy on write strategy [8].

Client The HDFS client is the access point to the filesystem. It is the one that provides transparent data access for users encapsulating requests to be sent to namenode though the network, hiding errors by selecting a different block replica to read, and offering a POSIX-like API. The client is responsible for coordinating the data flow between the user and datanodes, calculating the block's checksums when writing and verifying them when reading. The HDFS client uses local files for client side cache in order to improve performance and throughput.

2.2 Data Integrity

To protect itself against data corruption, HDFS make an intensive use of checksums. When a file is created, the checksum for each section of each block is computed and stored in the file that holds the block's metadata. Since the default section size is 512 bytes and CRC-32 uses 4 bytes, the storage overhead is less than 1%.

When a client reads from the datanodes, the checksums for the sections are computed and verified against the ones stored within the datanode. If the checksums do not match, the client notifies the namenode. The namenode marks the current block replica as corrupt, redirect clients to other datanodes and schedules a healthy copy of the current block to be re-replicated in another node. Once the replication for the block returns to a normal level, the degenerated copy is deleted.

2.3 Replication

The main mechanism of HDFS to deliver fault tolerance and high availability of data is replication. With replication, HDFS can server a read request selecting the closest copy to client, reducing network usage and achieving higher throughput. It also helps to cope with a great read demand for a file, because clients can be redirect to different datanodes holding the same file. In case of crash of current servicing datanode or of a corrupted block, another replica can be used to satisfy a client requisition.

The default replication factor is three, albeit it can be redefined for each file independently. The block reports sent by datanodes allow the namenode to have a global view of block availability and to take further actions to maintain the replication at normal levels for all files.

Replica placement is a tricky problem. As we are going to see soon on subsection 2.5, in a write operation, the datanodes for which the block is going to be replicated are arranged into a pipeline. So, the HDFS replica placement algorithm is to place the first replica in the same node as of the writer client (if the client is not running in a datanode, a datanode is chosen at random), the second one on a different rack, and the third one on a node of the same rack as the second as shown on Figure 2. The remaining replicas are placed at random nodes, trying to keep an evenly distribution. This confers to the system a good trade-off among reliability (replicas are placed on two different racks at least), write bandwidth (data has to traverse only one switch), read performance (there is two racks to chose to read from), and replica distribution.

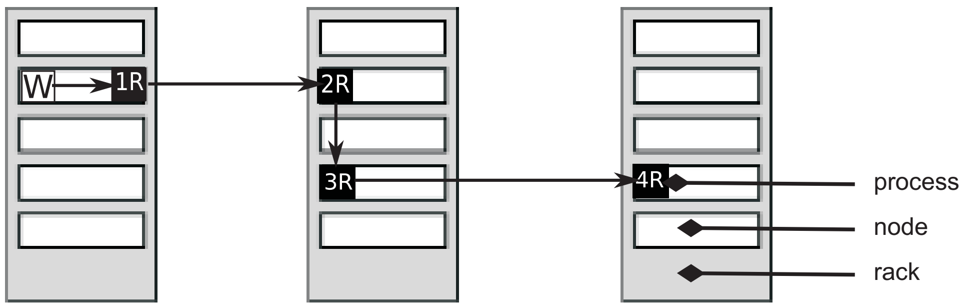

Figure 2: The replica placement on HDFS.(*Adapted from [28]*)

2.4 Coherency Model

The POSIX standard say that after a successful write to a regular file "*any successful* `read` *from each byte position in the file that was modified by that* `write` *shall return the data specified by the* `write` *for that position until such byte positions are again modified*" [53]. HDFS relaxes that POSIX imposition in favor of performance.

In the coherency model adopted by HDFS, a file is visible to any other reader after it has just been created. With regard to the file contents, all blocks are visible to other readers except by the one currently being written. If the HDFS client fails, the data stored on that block is lost.

In order to prevent that, the client shall invoke the `sync` operation at suitable points. The `sync` operation forces all buffers to be synchronized to datanodes. That means that all data written to the file so far is persisted and visible to other readers.

HDFS embraces the many-readers-single-writer model. That choice simplifies the implementation of HDFS with regard to data consistency and coherence.

2.5 Filesystems Operations

If a client wants to create a file, it sends a request to the namenode. The namenode checks if the file does not exist yet and if the client has sufficient permissions. If so, the namenode returns to the client an ordered list of datanodes to store the block replicas. So, the write operation occurs in a pipelined fashion, with one datanode forwarding data to the next datanode in the chain. If one datanode fails to write the data, it is removed from the pipeline and the write continues with the remaining nodes. The namenode schedules the re-replication of current and next blocks to be executed later. A write succeeds as longs as the minimum replication is maintained.

For read requests, the namenode returns a list of datanodes containing the block replicas. The list is ordered by the proximity of datanode to the client. The clients reads from the first datanode in the list. If the client finds a corrupt block, it warns the namenode, so it can trigger the necessary actions to restore the copy. The client then continues reading from the next datanode in the list. A read will fail only if no healthy replica is found.

The metric used by HDFS for distance is the bandwidth because it is the principal limiting factor for performance. The larger the bandwidth is between two nodes, the closer they are. Because bandwidth is a hard thing to measure, Hadoop makes an approximation assuming that bandwidth between two nodes will decrease in the following sequence as show on Figure 3:

- nodes are the same (*d1*).
- nodes are in the same rack (*d2*).
- nodes are in different racks of the same datacenter (*d3*).
- nodes are in different datacenters (*d4*).

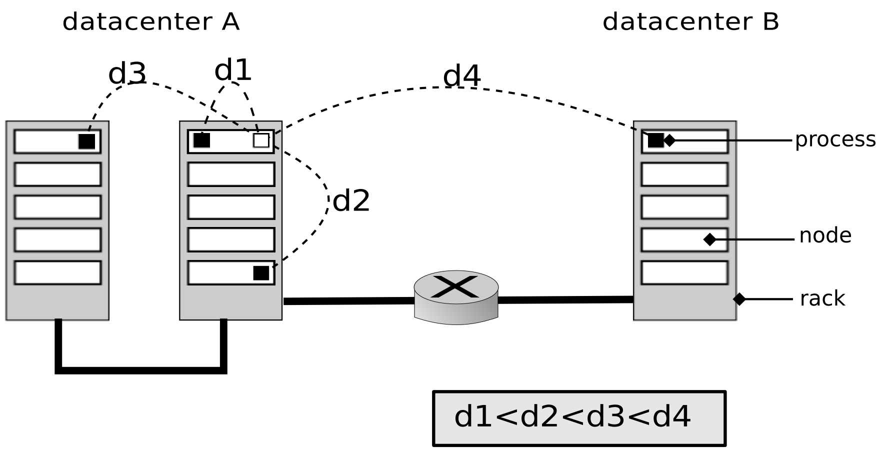

Figure 3: The relation among distances between two nodes in Hadoop (*Adapted from [28]*)

Regarding deletion, HDFS has a trash facility. Deleted files are not immediately removed, but placed in a trash folder and stay there for a user-defined amount of time before being garbage-collected.

3 High Availability in HDFS

Despite its design simplicity, the Hadoop Distributed Filesystem is very powerful. It can store petabytes of data and still provide the throughput required by high-performance computing applications. After detailing the inner workings of HDFS, we are ready to focus the discussion on the high availability problem.

So far, the high availability of data seems to be a solved problem. It is achieved on HDFS through block replication. The replica placement algorithm of HDFS ensures that no two replicas of the same block are located on the same node and that a block is replicated to at least two racks. So if a node goes down, data can still be read from another node. The scenario in which the entire rack is damaged is still no big deal. In that situation, the client just switches to the other rack and continues the work. If a user foresees a great read demand for a file, he or she can set a higher replication factor for it. That enables several readers to work in parallel, getting their data from different replicas. Therefore the impossibility of executing a single data operation is a rare condition on HDFS.

On the other hand, the overall system availability is something that needs to be improved. The single master approach taken by HDFS concentrates all the namespace information to just one point, the namenode. The namenode makes use of checkpoints and transactional logs stored on multiple locations to have that information persisted. However, that just prevents all files from being completely lost because otherwise nowhere else would have the information of which blocks belong to each file or even which files are stored in the system. If the namenode is recovering from a crash, it has to wait for block reports from almost all datanodes before being able to properly process client requests. That wait may take more than half hour for large clusters [8]. Besides that, maintenances or upgrades of namenode require the entire filesystem to be brought down.

It is easy to see now that the namenode is a single point of failure for HDFS requiring a more reliable machine compared to any other node, which violates the goal of using commodity hardware. As a matter of fact, there are some on going efforts to solve the problem or at least create the infrastructure for future solutions. But before we start detailing those efforts, it is useful to characterize the namenode runtime state.

Adapting the jargon introduced by [12], the namenode runtime state can be divided in two: the *hard state* and the *soft state*. The hard state consists of the namespace information (the filesystem tree, permissions, ownership, access and modification times, quotas, inode to blocks mapping, etc.) and it is the only state that is currently persisted in the form of checkpoints and logs. The hard state is minimal and stable. Minimal in the sense it contains the minimum information required to bring namenode to an operational state. Stable because between a failure and a recovery, the state does not change as it only depends on the namenode itself.

The soft state is composed by lease information, block replica locations, and replication status. State related to replication depends on datanodes, which may fail at considerable rates (2-3 failures per thousand nodes per day [22]). Lease state depends on namenode and clients and it is valid for a small fixed period of time. Thus, soft state is very volatile to be saved and reused later for recovering [17]. A better recovery strategy is to replicate the soft state to order nodes, so the system will always have an up-to-date copy of it in a live node.

Based on whether the hard and soft states are backed up or actively replicated, it is possible to classify a solution for high availability on HDFS, adapting from the categories described by Shvachko in [51]:

1. **Passive Replication** A node, the master, takes a leadership role being responsible for processing client requests. Remaining nodes stay on a standby position, assuming the master role by election in case the current one crashes. Based on how much state information is feed to standby nodes during master's runtime, we can define subcategories:

    (a) **Cold standby** Hard state is saved to stable storage. In the event of a failure, a new namenode process has to start from the backed up hard state and to recreate the soft state.

    (b) **Warm standby** Hard state is constantly replicated to other nodes. The new master, during takeover, has to reconstruct the soft state.

    (c) **Hot standby** Both hard state and soft state are pushed to the standby nodes by master. In this case, the new master can handle any client request as soon it takes over, since it has a fresh copy of the entire state.

2. **Active replication** The nodes are implemented like deterministic state machines, i.e., given the same sequence of input will all reach the same state. So, in this configuration, block reports, heartbeats, and client requests are sent to all nodes in the same relative order.

3.1 Passive Replication

In passive replication [13, 18, 23], also know as primary-backup, only one node, the master or primary, is allowed to process client requests. The master is also the responsible for sending an up-to-date state to the other replicas (the slaves or backups). Since just one node handles all the requisitions, a global ordering of operations is easily established. So all replicas agree on the same state.

If the master is detected to be out of work, an election takes place and one slave is promoted to master. The period of time between the crash of the previous master and the take over the new one is called *failover*. During failover a client request may fail. The warmness of the replication, i.e., the fraction of whole system state that is present on main memory of the replicated server, determines the failover time. That happens because the colder the replication is, the longer will be time spent on state reconstruction from checkpoints, logs, or other sources.

Passive replication does not need the replicas to behave deterministically if a state copy approach is used for replication. Thus, in that case, it can take full advantage of multithreading without any special scheduling control. The model requires $t + 1$ replicas to be t-fault tolerant. Byzantine failures are not supported. One possible optimization to this approach is to allow clients to send read-only request to backups. In the following subsections, we present some solutions for high availability in HDFS in a increasing order of warmness.

3.1.1 Cold standby

DRDB and Heartbeat ContextWeb[1], a web advertisement company, combined DRDB from LINBIT with HeartBeat from Linux-HA to provide a cold-standby solution for high-availability of the namenode [2, 16].

DRDB stands for Distributed Replicated Block Device and *"can be understood as network based RAID-1"* [33]. DRDB is implemented as a block device such that any data written to it is mirrored over a TCP connection to another storage device, thus yielding transparent data replication for applications. And because the mirroring occurs at block level, DRDB can be used with any filesystem. DRDB 8.3, the current release, only supports two nodes. They can both work as a primary-backup pair and also as an active-active pair if used with a shared cluster file system with distributed lock manager such as GFS2[2] and OCFS2[3]. If a faulty node recovers, DRDB takes cares of synchronizing it with the active one. It also can cope with split-brain scenarios – when due to link or software failures both nodes think they are the primary – employing a user defined strategy to reconciliate the replicas.

Heartbeat [41] is part of Linux HA project and provides cluster membership and messaging services. In other words, Heartbeat can tell which nodes belong to the cluster, which are alive and provide means for them to exchange messages. Heartbeat also used to be a cluster resource manager (CRM), but recently the CRM module forked into its own project called Pacemaker [48]. The CRM is the piece of software responsible for ensuring the high availability of a cluster by taking actions like starting services and processes on other nodes. Pacemaker can support different cluster configurations from primary-backup to multiple active and passive nodes.

ContextWeb's proposal uses a primary and a backup namenode. Each node is connected to a different switch so if a switch is damaged just one namenode is affected. All nodes (including namenodes and datanodes) are connected to switches by two physical links that are bonded with LACP/802.3ad [45] to work as a single virtual link. The virtual link delivers a higher aggregated bandwidth at the same time it provides some redundancy to make the solution resilient to link failures. In this solution, DRDB is used to replicate the checkpoint and the transactional log to the backup namenode while HeartBeat monitors the primary namenode and performs failover procedures. A Virtual IP address is used, so datanodes and clients do not need to be aware of which namenode they are talking to.

If Heartbeat detects the crash of the primary namenode, it will do the failover by making the Virtual IP to point to the backup namenode, promoting it to primary in DRDB and starting the namenode process on it from the node's copy of the checkpoint and transactional log. All this process will take about 15 seconds to complete, although a fully operational namenode will take longer since it has to wait for block reports from almost all datanodes to recreate the soft state and exit safe mode. Regarding leasing, nothing special is done so clients will have to recover their leases in order to continue to write.

This solution is 1-fault tolerant and capable of handling failstop crashes and link failures

[1]`http://www.contextweb.com/`
[2]Global Filesystem 2. Not to be confused with the Google File System
[3]Oracle Cluster Filesystem 2

due its redundant network infrastructure. It falls into the cold standby category because the backup namenode is not kept running and fed with state information while the primary is alive. If Pacemaker were used as the CRM, it might bring the ability to failover to more than one node. However, the restriction of two nodes for DRDB limits the fault tolerance of this solution. So ContextWeb developers are considering other replication methods like GlusterFS, RedHat GFS or SAN/iSCSI backed. If failover time is not a problem, this is a quite handy solution for high availability since no modification to stock HDFS is needed.

3.1.2 Warm standby

Backup node and standby node The backup node proposed by Shvachko [50] and introduced in Hadoop 0.21 implements a more efficient checkpoint strategy than current secondary namenode because it does not have to retrieve the last *fsimage* and *edits* from namenode. Instead it keeps its own namespace view in sync with namenode.

In the Java programming language, all I/O is abstracted as streams. In the implementation of backup node, a new special stream is created, so the operations on namespace are recorded to it as if it were an additional storage area for the transactional log. But this new stream actually propagates the operations to a registered backup node to the namenode. The backup node can then apply the operations to its own in-memory data structures and record to its own transactional log just like the namenode (in fact, at implementation level, the backup node is a subclass of namenode). Thus the backup node keeps an up-to-date view of the filesystem metadata and it has all the information needed to create a new checkpoint by itself. While performing a checkpoint, the backup node cannot process the filesystem operations in real time. The problem is solved by recording the operations to a spool, so the node can catch up with them latter. The current HDFS implementation allows only one backup node.

Since it has a fresh view of namespace, the backup node is a serious candidate to a warm standby solution. As a matter of fact, that was the long term goal of Shvachko's proposal. In this new configuration called standby node, a failover mechanism would allow the node to take over a crashed namenode. There is also a possibility of supporting multiple standby nodes. However, making the namenode forwarding the operations to many nodes would require the global HDFS lock to be held longer, compromising the system performance. If, instead, this redirection were done asynchronously by a background thread, the operations would have to be buffered, which would increase the memory pressure on the namenode [51].

A standby node solution with t standby nodes would offer t-fault tolerance for fail-stop failures. Until this moment, there is no implementation of the standby node strategy incorporated to the HDFS baseline.

Hadoop High Availability through Metadata Replication Researchers of IBM Research developed a solution that may employ two architectures: an active-standby pair or a primary-slaves cluster. The main difference between the two is that the slaves of the latter one are allowed to reply to read-only requests. The researchers claim in [24] that both architectures are very suitable to read dominant scenarios with medium amount of files – the common case for MapReduce applications. Besides the file system metadata, the solution devised by them also replicates the lease information, which places the solution in the frontier between the warm and hot standby categories. The researchers justify that choice by stating that a full copy of master state would incur in a great performance penalty. Lease state is replicated to ensure the consistency of write operations, so a still valid lease is not lost after failover. The lifecycle of the approach is divided in three phases: initialization, replication, and failover. We assume that the active-standby pair can be viewed as a particular case of primary-slaves architecture where an election to define the new master is not needed. Based on that assumption, the remaining of this description will refer to the primary-slaves architecture except where noted otherwise.

In the initialization phase, the primary waits for the slaves to register to it. When a slave does so, the primary broadcast a table containing IP addresses for all the already registered slave nodes, so they can be aware of their peers and be able to organize elections in the event of the primary crashes. After all nodes are registered, they check with the primary whether they have a copy the latest checkpoint of the filesystem. If a node does not have it, the primary will send it a copy. When all nodes are synchronized to the primary, the system can proceed to the next phase.

The second phase, replication, is the basis for high availability. Filesystem operations and lease requests are intercepted at the primary and replicated to slaves. The way primary and slaves are kept in sync is determined by an adaptive online decision module. The module compares the average network throughput and workload performance against user defined thresholds in order to decide which of three synchronization modes to use. That flexibility helps the system to meet the performance requirements for different workloads. The three possible synchronization modes are the following:

- **Mode 1:** The primary persists metadata operations on disk before sending them to slaves. A client request is committed after slaves have also persisted the operation and sent the acknowledgment to the primary. This mode is recommended for LANs with high bandwidth.

- **Mode 2:** The primary persists metadata operations on disk before sending them to slaves. Slave nodes send an acknowledgment to the primary right after they receive the operation. When the master receives the acknowledgment messages, the client request is committed. This mode is also recommended for LANs with high bandwidth. The choice between modes 1 and 2 depends on how much the user is willing to sacrifice the synchronization protection in favor of performance. Mode 1 imposes bigger latency than mode 2, but in the latter slaves might fail to write the metadata operation to disk, generating inconsistencies.

- **Mode 3:** The primary sends metadata operations to slaves before persisting them to disk. Slaves send acknowledgment messages to the master before storing the operation on disk. The client request is committed after the primary stores the metadata operation on disk and receives the acknowledgements. This is the best mode for WANs with low bandwidth because it offers better metadata transfer performance. The developers of this solution did not comment it, but this mode has the potential pitfall of allowing slaves to store the metadata for a non-committed request when the primary crashes before writing the operation to disk.

To ensure the consistency among nodes, a non-blocking three-phase commit protocol is employed. But to avoid performance degradation, the primary commits the request after receiving acknowledgements from a majority of nodes. Besides acknowledgment messages, slaves send heartbeats to primary to tell they are still alive. The primary, in its turn, also sends control messages regarding which synchronization mode slaves should use.

The third and last phase is failover. If some slave thinks the primary is down, it starts an election by generating a unique sequence number and sending it to all other slaves. That sequence number must be higher than any other the node has seen. The uniqueness of a sequence number sn is guaranteed because it must be such $sn \equiv id \pmod{n}$, where n is the number of slaves and id is the node's id. Both information are available on the IP table received at the initialization phase. A slave that receives a sequence number may send back a disagreement message if it believes the master is still running or if it saw a bigger sequence number. Otherwise it sends an agreement message. A node that receives agreement messages from a majority becomes the new primary and broadcasts a leadership announcement to all nodes. In order to make failover transparent to datanodes and clients, the new primary transfer to itself the IP of the previous one. Once the transfer is done, the primary requests all slave nodes to re-register like on phase one and then it starts reconstructing the block to datanode mapping.

To evaluate the efficiency of solution, the researchers of IBM China used a 5 node cluster composed by Pentium 4 machines with 1.5 Gb of DDR RAM and connected by 1Gbps Ethernet links. One node played the role of the active, a second node played the standby and the remaining played the datanodes. They made several experiments varying the number of one-block files from 5,000 to 100,000. Because of the cluster configuration, high bandwidth, synchronization mode 1 was used all the time. To test failover they simply unplug actives's power cord and to test metadata replication they just created a file.

It was observed that failover time is proportional to number of blocks in the filesystem and that the time spent on network transfer was noticeable higher than the time spent to process block reports. For 100,000 blocks a failover time of no much more than seven seconds was achieved.

Regarding the cost of replication for the processing of filesystem metadata, again network transfer time had the biggest impact, accounting for half of the overall time spent for replication. The number of blocks did not seem to affect performance, since a metadata operation record has a constant size. The overall result is that the time to process filesystem metadata has doubled if compared to regular HDFS.

In conclusion, IBM Research's solution offers t-fault tolerance for failstop failures for a cluster of $t + 1$ nodes. The solution has the limitation that any new or recovering node can only join the cluster at the initialization and failover phases. Since Hadoop clusters tend to be a lot larger, the solution should undergo tests on environments with higher number of nodes to verify if it still behave well and if synchronization modes 2 and 3 will not cause state corruption at slaves.

3.1.3 Hot standby

Avatar Nodes The avatar node strategy, outlined by Borthakur in [5], was developed at Facebook to avoid a downtime of about an hour in a 1200-node and 12-petabyte HDFS cluster when applying code patches to namenode. The cluster ran Hadoop 0.20, which does not have the backup node feature. The developers did not want to modify or upgrade the system code because that would mean long hours of testing before deploying it in production. The solution found was to create a wrapper layer around existing namenode and datanodes to achieve high availability.

In this proposal there is a primary avatar node and a standby avatar node. The primary avatar node behaves exactly like the namenode. In fact, it runs the same code and writes the transactional log to an NFS share. The standby avatar node encloses one instance of namenode and one instance of secondary namenode. The latter is still the one responsible for generating checkpoints. The former is kept on safe mode, so it will not process any requests. The standby avatar node constantly feeds it with operations retrieved from the log in the NFS share. Datanodes are also encapsulated by avatar datanodes. Avatar datanodes forward block reports to both primary and standby avatars. This way, the standby avatar namenode has all the information needed to act as a hot standby.

The failover procedure for this approach takes few seconds to complete and is triggered by the administrator via a command line program. Because it is a manual process, the administrator can ensure that the control is switched to the standby avatar only when it is guaranteed that the primary is really down, therefore preventing a split-brain scenario. Before assuming the primary role, the standby avatar makes sure it has processed all pending transactions.

In other to make failover transparent to clients, they reach the avatar node in charge via a Virtual IP. Clients reading from HDFS will not be affected by failover, because they cache the location of block replicas. Thus they do not need to contact the primary node during failover. On the other hand, because new block allocations are not recorded until the file is closed, writter clients will observe failures for their writes during a failover. Most applications can tolerate this behavior. For instance, Hadoop MapReduce reschedule any failed tasks and HBase always sync/flush the file contents before completing a transaction.

Although simple and little intrusive to code, this solution does not offer protection against crashes because it lacks some automatic failure detection and failover features. It is only capable of keeping the filesystem running during a programmed maintenance for which the shutdown of namenode is required.

Namenode Cluster In the Namenode Cluster approach [25] designed by China Mobile Research Institute, a cluster of nodes in a master-slaves scheme replaces the namenode. The master node of that cluster handles all client requests and propagates filesystem updates to slaves as well. It does the propagation with the help of the SyncMaster Agent, a daemon, which is also responsible for managing the slaves. Conversely, the slave nodes run each one an instance of the SyncSlave Agent to receive the updates and sync their data structures.

When a new namenode joins the cluster, it has to register to the master. The master will then block all metadata write operations and transfer state information to node to let it in a synchronized state. In the end of registration process, the recently joined slave is ready to begin to receive the updates.

All update transfer is done in units named SyncUnits. The SyncUnits carry, besides the data already found in the transactional log, data about lease operations, replication commands, block reports, and datanode heartbeats. Thus slaves have a copy of the entire state of the master. In order to reduce the load, information that came from the several datanodes are grouped and sent together to the slaves. The SyncUnits are sent to all nodes in parallel using separated threads. If some thread fails to send or times out, its related slave is marked as unsynced and it will have to register again. The SynUnits are also transferred in order to keep consistency. That means that a new SyncUnit is not sent to slaves until the completion of the transference of the current one. To avoid jeopardizing the master's performance, the updates are buffered. The larger the buffer, the greater the performance at memory expense.

Like in the ContextWeb's solution, failover is managed and performed by HeartBeat. In order to make the slaves, datanodes, and clients able to localize the master node, the information regarding which nodes belong to the namenode cluster is kept on Zookeeper [28, 30]. Zookeeper is a highly-available distributed coordination service, also developed under the umbrella of the Hadoop project. Zookeeper keeps a replicated tree of nodes that can store up to 1 megabyte each. All operations like node creation, node deletion, and data storage are atomic. Users of Zookeeper can set watchers on a node to be informed of changes on it or in any of its children. Leaf nodes can be created as ephemeral, i.e., they exist while the user session is still active, providing some liveness detection. Besides that, Zookeeper can be told to add sequence numbers to the name of newly created nodes, which can be used to define a global ordering. All those features provide the basic constructs to implement several distributed protocols like membership management, atomic broadcast, leader election, distributed data structures, distributed lock, etc. Future works on namenode cluster involve the use of Zookeeper for failover.

For this solution, it was observed a 15% of performance reduction for typical file I/O operations and Hadoop MapReduce jobs and 85% of performance reduction for metadata write operations like `touch` and `mkdir` if compared to vanilla HDFS. The I/O operations were not heavily impacted because they require little interaction with the namenode. On the other hand, metadata write operations became a great burden due to the synchronization process.

In conclusion, the namenode cluster yields t-fault tolerance for a cluster of t namenodes but it might not be the best solution for environments where changes to filesystem metadata are dominant. The code for the Namenode Cluster is currently available at `http://github.com/gnawux/hadoop-cmri`.[4]

3.2 Active Replication

Active replication or state machine replication [13, 18, 23, 21] does not impose any hierarchy among nodes. Clients send requests to all replicas via atomic multicasts. Each replica

[4]Last access on August 4th, 2010.

executes the request and replies to the client, which may pick the first answer or wait for a majority of identical answers if it desires protection against Byzantine faults.

For active replication to work, the replicas must reach a consensus about the current state. For that reason, they are implemented as deterministic state machines, i.e., they all end up in the same state if the same sequence of inputs is supplied to them. Additionally, all the replicas must agree on the order that the operations should be applied. Therefore the replication protocol must enforce that requirement. The order can be relaxed if operations are idempotent or commutative.

It is important to notice that there is no failover for active replication. If some node fails, the system can still make progress with the remaining nodes. Clients will not even notice the problem, since all other correct nodes will execute the request and deliver the reply. A node that recovers must catch up its state with the other replicas before processing any requisition.

This approach can support failstop failures and Byzantine failures as well. For the first failure model, a t-fault tolerant system requires $t + 1$ replicas. For the latter model, $2t + 1$ replicas are required. In doing so, we always have a majority of at least $t + 1$ votes for a certain reply. So far, there is only one example of solution that explores this technique:

UpRight-HDFS The major goal of the UpRight library [12] is to make easy to implement Byzantine Fault Tolerance (BFT) into existent Crash Fault Tolerant (CFT) systems at a low additional reduction on performance. The Upright library conceptually divides the space of Byzantine faults into omission failures (node stop sending messages) and commission failures (node sends messages not specified by the protocol). The name UpRight comes from the promise that an UpRight service can still run with correctness (right) despite of r commission failures and any number of omission failures; and can still be live (up) if it presents at most u failures, out of which up to r failures are commission failures, during long enough synchronous intervals. The parameters u and r are defined by user and determine the required size for an UpRight cluster.

UpRight employs state machine replication. Clients and servers see the library as a communication proxy, in which clients send requests through the client version of the library and servers send replies through the server version of library. The messages are redirected to the UpRight cluster, which is responsible for validating and authenticating the messages, ordering the requests and atomically broadcasting them to the many deterministic server instances, establishing the consensus on server replies, commanding servers to checkpoint their state, among many other services required for a BFT system to work.

To prove they have reached their goal, the developers of UpRight have modified HDFS to use their library [12, 40]. The integration of the UpRight library demanded changes to less than 1750 lines of the namenode's code. Most of the changes were related to checkpoint management and generation, because now the whole state of the namenode would have to be recorded. They also spent a considerable amount of time removing non-determinisms from the execution path of namenode. For instance, code that relied on random numbers, local system time or background threads had to be modified. So, they made all servers to agree on the same random seed, to use logical time provided by UpRight and to let

UpRight to schedule background tasks. The transactional log maintained by the namenode was disabled, since UpRight has its own logging system.

HDFS client also had to be changed to send requests through the UpRight client library. The library makes transparent the fact that the client is actually sending the requests to a cluster. Furthermore, it also makes the client unaware of faulty namenodes. The developers also worked on the datanodes in order to make corrupted blocks detectable even if the datanode supplies a correct checksum for them. For that, they stored on the namenode a cryptographic hash for each block, so the hash can be checked against the datanode when a block is read from it.

For the evaluation of UpRight HDFS they used 107 Amazon EC2 small instances, 50 datanodes, 50 clients and a block replication factor of 3. In addition to UpRight HDFS configurations with $u = r = 1$ and $u = 1$, $r = 0$ (in fact, an Upright service behaving like a CFT system); stock HDFS was also tested for comparison. The read throughput did not present much deviation from HDFS even for $r = 1$. Regarding write throughput, Upright HDFS with $r = 0$ showed a reduction of less 20%, while the configuration with $r = 1$ presented 70% of performance reduction due apparently to the time to reach consensus on the replicated namenodes. Regarding CPU consumption, UpRight HDFS can consume up to a factor of 2.5 more CPU when writing. Those costs are compensated by the fact that if a namenode crashes and it has its checkpoint corrupted, the system can still make progress, while in standard HDFS the entire filesystem would be lost. Moreover, clients do not even notice the incident.

Current HDFS does not spend great efforts on enforcing security, which makes protection against Byzantine faults somewhat pointless. However the experience acquired with UpRight HDFS can help to construct other high availability solutions using state machine replication. Code for the UpRight library and UpRight HDFS is published at `http://code.google.com/p/upright/`[5].

3.3 Recovery

So far we have discussed only one aspect of high availability: fault tolerance. The other aspect is recovery. Once the namenode crashes it must be able to recover to a consistent state. We have already introduced the recovery strategy of HDFS: a combination of checkpoints performed by a helper process and pessimistic message logging. That approach is efficient because namenode is not blocked during checkpointing, checkpoints are created less frequently than they would be in conventional techniques, and recovery to the very last correct state is possible since all issued operations since last checkpoint are recorded to a stable storage.

The meaning of stable storage for HDFS is to store the checkpoints and logs to several disks. To provide some protection against severe damages on the namenode's machine, they are also kept on remote storages through NFS mounts. As a last resource, an administrator can use the checkpoints kept by the secondary namenode, although they may not represent the most up-to-date state of the filesystem.

[5]Last access at August 4th, 2010.

An effort to provide a safer storage for the transactional log is the integration of BookKeeper with HDFS[44]. BookKeeper[32] is a reliable and highly available replicated service for write-ahead logging. In BookKeeper, a client add entries (records) to a *ledger* that is stored in a *ensemble of bookies*, the replicated servers. To achieve high throughput for reads and writes, the ledger entries are stripped, i.e., they are written to a subset of bookies in the presence of an available quorum. BookKeeper uses Zookeeper to distribute information about ledgers and available bookies to clients.

The integration of BookKeeper follows a similar approach to the one taken by the backup node. A new stream abstraction is created to store the transactional log, so it is transparent to the namenode whether the filesystem operations are being recorded to the BookKeeper ensemble or to a file. The integration is still a work in progress and it is expected to deliver *"higher throughput performance in large deployments"* and *"higher availability through externally recoverable log files"*[44].

4 High Availability in other Distributed Filesystems

In this section, we expose how other distributed filesystems deal with failures. They are all targeted for HPC and/or large storage. They are also based on the concept of keeping filesystem namespace orthogonal to data.

It seems there is a consolidate nomenclature for such systems: inode-like structures are managed by a metadata server (MDS) and data is stored on several object storage devices (OSDs). Files are stripped into objects. In order to keep generality, we stick to those terms. For instance, for HDFS the MDS is the namenode, the OSDs are the datanodes, and the objects are the blocks.

GFS Like in its pupil, GFS [20] replicates objects to at least three OSDs. The MDS uses transactional log and checkpoints, which are replicated to several machines. A change on MDS state is only committed when flushed to disk and copied to all replicas. In case of failure of the MDS, a new one is started anywhere from the transactional log and checkpoint. Besides the main MDS, there are also *"shadow masters"* providing read only access to filesystem and improving read performance. Since they feed themselves from a copy of the log they may lag behind the current MDS state.

Lustre Lustre [46] relies on RAID technology to protect data. Lustre's MDS is replicated in an active-standby fashion, with both replicas using a shared storage. Lustre also makes use of transactional log. There are plans for release 2.2 to use several MDSs, each one responsible for a disjoint sub-tree of the filesystem.

Ceph Ceph [26] uses RAID, but also acknowledges the GFS motto "*failures are the norm rather than exception*". Thus, it makes use of object replication like HDFS and GFS. The filesystem tree is dynamically partitioned among many MDSs targeting an evenly access of MDSs by clients. Hot spots can be replicated to many MDSs in order to balance the load.

The journaling feature is absent, but it is on the roadmap considering that another node could take over a faulty MDS from the logs.

Panasas Regarding data fault tolerance, Panasa's [27] strategy is totally based on RAID. The namespace is statically split into volumes, each one managed by an MDS. The metadata server has a failover configuration such that the transactional log can be also forwarded to a remote peer. When the primary fails, a backup can assume the activities by reading the log replica.

Apple Xsan2 Apple's distributed filesystem Xsan2 [31] also employs RAID storage and journaling. The filesystem tree can be split into volumes, each one controlled by its own MDS. MDSs may have multiple standby MDSs with different failover priorities assigned for them. So, when a MDS goes down, the standby with highest priority takes over. The MDS and the RAID arrays are connected by dedicated redundant fibre channel paths to protect from link failures.

Most of distributed filesystems cited here trust on RAID schemes to protect data from loss and corruption. But, as denoted by Tom White [28], the replication strategy adopted by HDFS has some advantages over RAID. RAID 0 provides the best performance but it is still worse than replication. The reason for that is that for RAID 0 the slowest disk in the array limits the throughput. Whereas in the HDFS method, the disk operations are independent, so their speed is, in average, better than the slowest disk. Another argument in favor of replication is that the failure of a single disk in RAID schemes makes the whole disk array to become unavailable. For HDFS, while there are enough copies of blocks, the damaged disk is not a big problem. Replication also helps to cope with high demand for files because several copies can be read in parallel. The reader may have heard of HDFS-RAID [6, 39]. The main goal of that initiative is to save storage space by being able to using only two replicas, but still keeping the effective replication at factor of three. That is accomplished using erasure codes. The reported savings varies from 25% to 30%.

In general, the mentioned filesystems spend little effort to improve the high availability of the MDS. They tend to use cold standby techniques. In particular, Lustre, Xsan2 and Panasas employ the active-standby approach and journalling for recovering. They, together with Ceph, are based on the assumption that a partitioned filesystem tree among many MDSs may restrict the extension of crashes. However, all those strategies are not enough to prevent the system from becoming unavailable for a considerable amount of time.

5 Conclusion and Future Works

In this technical report we presented HDFS, a very scalable and high performance distributed filesystem. We started by providing a concise description of the system, outlining its architecture, elements, operations, and strategies toward performance, reliability, consistence, and data availability. That introduction gave us the foundations to present the high availability issue of HDFS, which is basically concerned with the fact that the namenode is a single point of failure. Presented the problem, we characterized the internal state of namenode into hard and soft states. That empowered the categorization of solutions, which was based on the replication approach taken and whether the hard or soft states were replicated. So we could make a qualitative exposition of proposals to solve or at least alleviate the problem in a logical order to the reader. Besides that, this technical report did not restrict high availability to fault tolerance, but also extended the discussion towards recovery.

Most of the solutions presented here trust on passive replication and a single standby node. That works well in batch environments, where failures can be tolerated with rescheduling of jobs and one backup node is enough to keep the system running during maintenances. Interactive environments require more robustness. IBM China's, Namenode Cluster and UpRight HDFS proposals can survive to multiple crashes of namenode, with the last one being able to handle Byzantine faults in a transparent fashion to clients. Despite all these efforts there is no definitive solution integrated into HDFS's code baseline yet. Either because the code is unavailable, or still in development, or even because it was based on an old version.

If we compare HDFS to other distributed filesystems, they are about in the same evolutional step with respect to high availability. The filesystems mentioned in section 4 limit themselves to standby MDSs, partitioned filesystem trees, and journaling which is essentially the same thing as the checkpoint and transactional logs of HDFS. Therefore, a contribution to high availability on HDFS means a contribution to high availability on distributed filesystems in general.

Our long-term goal is to implement a complete solution that could survive to a considerable number of the failures at low cost for performance, and to contribute it back to the Hadoop community. Considerable research has been done about or using Hadoop software. The ACM Digital Library (`http://portal.acm.org`) registered, until June 13th, 45 publications that have *hadoop* as keyword, 80 that have *mapreduce*, 8 that have *hbase*, and 4 that have *zookeeper*. And those results are only for publications of 2010. But little of them were incorporated back. The reason for that is that the work is generally made in isolation, with no contact with the Hadoop community, and using a frozen baseline for a prototypical quality code. We, instead, intend to work close to Hadoop developers, taking parting in the newly created Hadoop Enhancement Proposal (HEP) process [37]. In doing so, the theory and correctness of academy can meet the practice and experience of industry.

Initially we are betting on Zookeeper as the basis of our solution, since it provides ways to implement scalable atomic multicast, offers global ordering of events, and, furthermore, it is also part of the Hadoop project. So it can be extended or tuned for our needs. To prove the concept, we are working in a small prototype of a replicated server that can receive requests from clients to modify a single integer. The prototype uses Zookeeper to log operations and to notify backups of updates.

References

[1] Alvisi, L., and Marzullo, K. Message logging: Pessimistic, optimistic, causal, and

optimal. *IEEE Trans. Softw. Eng. 24*, 2 (1998), 149–159.

[2] Bisciglia, C. Hadoop HA configuration. `http://www.cloudera.com/blog/2009/07/hadoop-ha-configuration/`. Last access on August 4th, 2010.

[3] Bockelman, B. Using Hadoop as a grid storage element. *Journal of Physics: Conference Series 180*, 1 (2009), 012047.

[4] Borthakur, D. Hadoop & its usage at Facebook. `http://www.snia.org/events/storage-developer2009/presentations/keynotes/DhrubaBorthakur-Hadoop_File_System_Architecture_Facebook.pdf`. Presented at the Storage Developer Conference on September 15th 2009 at Santa Clara, CA, USA. Last access on August 4th, 2010.

[5] Borthakur, D. Hadoop avatarnode high availability. `http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html`. Last access on August 4th, 2010.

[6] Borthakur, D. HDFS and erasure codes (HDFS-RAID). `http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html`. Last access on August 4th, 2010.

[7] Borthakur, D. HDFS high availability. `http://hadoopblog.blogspot.com/2009/11/hdfs-high-availability.html`. Last access on August 4th, 2010.

[8] Borthakur, D. The high availability story for HDFS so far. `http://www.borthakur.com/ftp/hdfs_high_availability.pdf`. Presented at ApacheCon on November 5th 2009 at Oakland, CA, USA. Last access on August 4th, 2010.

[9] Borthakur, D., and Zengh, S. Development at Facebook Hive and HDFS. `http://www.slideshare.net/cloudera/hw09-hadoop-development-at-facebook-hive-and-hdfs`. Presented at Hadoop World on October 2nd 2009 at New York, NY, USA. Last access on August 4th, 2010.

[10] Cafarella, M., and Cutting, D. Building Nutch: Open source search. *Queue 2*, 2 (2004), 54–61.

[11] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 1–26.

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

[13] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[14] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[15] Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[16] Dorman, A., and George, P. Production deep dive with high availability. `http://www.scribd.com/doc/20971412/Hadoop-World-Production-Deep-Dive-with-High-Availability`. Presented at Hadoop World on October 2nd 2009 at New York, NY, USA. Last access on August 4th, 2010.

[17] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 29–43.

[18] Guerraoui, R., and Schiper, A. Software-based replication for fault tolerance. *Computer 30*, 4 (1997), 68–74.

[19] HDFS architecture. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`. Last access on August 4th, 2010.

[20] McKusick, M. K., and Quinlan, S. GFS: Evolution on fast-forward. *Queue 7*, 7 (2009), 10–20.

[21] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv. 22*, 4 (1990), 299–319.

[22] Shvachko, K. HDFS scability: the limits to growth. *;login: The Usenix Magazine 35*, 2 (April 2010), 6–16.

[23] Tanenbaum, A. S., and Steen, M. v. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[24] Wang, F., Qiu, J., Yang, J., Dong, B., Li, X., and Li, Y. Hadoop high availability through metadata replication. In *CloudDB '09: Proceeding of the first international workshop on Cloud data management* (New York, NY, USA, 2009), ACM, pp. 37–44.

[25] Wang, X. Pratice of namenode cluster for HDFS HA. `http://gnawux.info/hadoop/2010/01/pratice-of-namenode-cluster-for-hdfs-ha/`. Last access on August 4th, 2010.

[26] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. 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, 2006), USENIX Association, pp. 307–320.

[27] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. 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, 2008), USENIX Association, pp. 1–17.

[28] White, T. *Hadoop: The Definitive Guide*. O'Reilly Media Inc, Sebastopol, CA, June 2009.

[29] Apache Hadoop. `http://hadoop.apache.org/`. Last access on August 4th, 2010.

[30] Apache Zookeeper. `http://hadoop.apache.org/zookeeper/`. Last access on August 4th, 2010.

[31] Apple Xsan2. `http://images.apple.com/xsan/docs/L363053A_Xsan2_TO.pdf`. Last access on August 4th, 2010.

[32] BookKeeper overview. `http://hadoop.apache.org/zookeeper/docs/r3.3.1/bookkeeperOverview.html`. Last access on August 4th, 2010.

[33] DRBD. `http://www.drbd.org`. Last access on August 4th, 2010.

[34] Facebook: facts & figures for 2010. `http://www.digitalbuzzblog.com/facebook-statistics-facts-figures-for-2010/`. Last access on August 4th, 2010.

[35] Facebook has the world's largest Hadoop cluster! `http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html`. Last access on August 4th, 2010.

[36] Google facts and figures (massive infographic). `http://royal.pingdom.com/2010/02/24/google-facts-and-figures-massive-infographic/`. Last access on August 4th, 2010.

[37] Hadoop Contributors Metting held on may 28th, 2010. `http://wiki.apache.org/hadoop/HadoopContributorsMeeting20100528`. Last access on August 4th, 2010.

[38] HBase. `http://hbase.apache.org/`. Last access on August 4th, 2010.

[39] Implement erasure coding as a layer on HDFS. `http://issues.apache.org/jira/browse/HDFS-503`. Last access on August 4th, 2010.

[40] HDFS UpRight overview. `http://code.google.com/p/upright/wiki/HDFSUpRightOverview`. Last access on August 4th, 2010.

[41] Heartbeat. `http://www.linux-ha.org/wiki/Heartbeat`. Last access on August 4th, 2010.

[42] Hot standby for namenode. `http://issues.apache.org/jira/browse/HDFS-976`. Last access on August 4th, 2010.

[43] Hunting disease origins with whole-genome sequencing. `http://www.technologyreview.com/biomedicine/24720/`. Last access on August 4th, 2010.

[44] Integration with BookKeeper logging system. `http://issues.apache.org/jira/browse/HDFS-234`. Last access on August 4th, 2010.

[45] Link aggregation. `http://en.wikipedia.org/wiki/Link_aggregation`. Last access on August 4th, 2010.

[46] Lustre. `http://www.lustre.org`. Last access on August 4th, 2010.

[47] NN availability - umbrella Jira. `http://issues.apache.org/jira/browse/HDFS-1064`. Last access on Agust 4th, 2010.

[48] Pacemaker. `http://clusterlabs.org/wiki/Main_Page`. Last access on August 4th, 2010.

[49] The Open Group Base Specifications issue 6. `http://www.opengroup.org/onlinepubs/000095399/`. Last access on August 4th, 2010.

[50] Streaming edits to a standby name-node. `http://issues.apache.org/jira/browse/HADOOP-4539`. Last access on August 4th, 2010.

[51] The namenode should forward block reports to backupnode. `http://issues.apache.org/jira/browse/HDFS-839`. Last access on August 4th, 2010.

[52] Worldwide LHC Computing Grid. `http://public.web.cern.ch/public/en/lhc/Computing-en.html`. Last access on August 4th, 2010.

[53] Write - The Open Group Base Specifications issue 6. `http://www.opengroup.org/onlinepubs/000095399/functions/write.html`. Last access on August 4th, 2010.