Fault Tolerance in Hadoop for Work Migration

Jared Evans
CSCI B534 Survey Paper
3/28/11

1. Objective

The objective of this survey paper is to examine the distributed software package Hadoop and take a detailed look at how it handles Fault Tolerance. How Hadoop's Fault Tolerance affects work migration and location will also be explored.

2. Introduction

Hadoop [1] is an open-source software framework implemented using Java and is designed to be used on large distributed systems. Hadoop is a project of the Apache Software Foundation and is a very popular software tool due, in part, to it being open-source. Yahoo! Has contributed to about 80% of the main core of Hadoop [3], but many other large technology organizations have used or are currently using Hadoop, such as, Facebook, Twitter, LinkedIn, and others [3].

The Hadoop framework is comprised of many different projects, but two of the main ones are the Hadoop Distributed File System (HDFS) and MapReduce. HDFS is designed to work with the MapReduce paradigm. This survey paper is focused around HDFS and how it was implemented to be very fault tolerant because fault tolerance is an essential part of modern day distributed systems.

3. Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is the file system component of the Hadoop framework. HDFS is designed and optimized to store data over a large amount of low-cost hardware in a distributed fashion [1]. HDFS is comparable to Google's BigTable [6].

HDFS is designed for a large amount of big data files. A typical data file stored using HDFS could range from gigabytes to terabytes in size [2]. HDFS can support millions of files and can scale to hundreds of nodes.

HDFS stores file system metadata and application data separately [3]. The HDFS metadata is stored on a dedicated server called the NameNode and application data are stored on other nodes called DataNodes which contain blocks of data that are usually just part of a particular file. The communication in HDFS among all of nodes in the system is done using a TCP-based protocol [3].

3.1 NameNodes

The NameNode records all of the metadata, attributes, and locations of files and data blocks in the DataNodes. The attributes it records are things like file permissions, file modification and access times, and namespace, which is a hierarchy of files and directories. The NameNode maps the namespace tree to file blocks in DataNodes. When a client node wants to read a file in the HDFS it first contacts the Namenode to receive the location of the data blocks associated with that file [3].

The NameNode stores information about the overall system because it is the master of the HDFS with the DataNodes being the slaves. It stores the image and journal logs of the system. The image of the system is a list of blocks and data for each file stored in the HDFS. The journal is just a modification log of the image. The NameNode must always store the most up to date image and journal. Basically, the NameNode always knows where the data blocks and replicates are for each file and it also knows where the free blocks are in the system so it keeps track of where future files can be written.

3.2 DataNodes

The DataNodes store the blocks and block replicas of the file system. During startup each DataNode connects and performs a handshake with the NameNode. The DataNode checks for the accurate namespace ID, and if not found then the DataNode automatically shuts down. New DataNodes can join the cluster by simply registering with the NameNode and receiving the namespace ID [3].

Each DataNode keeps track of a block report for the blocks in its node. Each DataNode sends its block report to the NameNode every hour so that the NameNode always has an up to date view of where block replicas are located in the cluster.

During the normal operation of the HDFS, each DataNode also sends a heartbeat to the NameNode every ten minutes so that the NameNode knows which DataNodes are operating correctly and are available. If after ten minutes the NameNode doesn't receive a heartbeat from a DataNode then the NameNode assumes that the DataNode is lost and begins creating replicas of that DataNode's lost blocks on other DataNodes. The nice thing about the HDFS architecture is that the NameNode doesn't have to reach out to the DataNodes, it instead waits for the DataNodes to send their block reports and heartbeats to it. The NameNode can receive thousands of DataNode's heartbeats every second and not adversely affect other NameNode operations [3].

3.3 Clients

The HDFS Client is a code library that allows user applications to access the interface of the HDFS instance. This allows client applications to read, write and delete files, and also create or delete directories.

When an application needs to read a file it first contacts the NameNode to receive a list of DataNodes that contain replicas of the data blocks it is looking for. Using that list, the client then accesses the appropriate DataNodes directly to read the contents of the file it is looking for. The list is also sorted by locations closest to the client node so as to minimize the communication overhead.

When a client needs to write a file to the HDFS it contacts the NameNode requesting a DataNode location where it can write the blocks of the file and it also requests locations to write the replicas of the file.

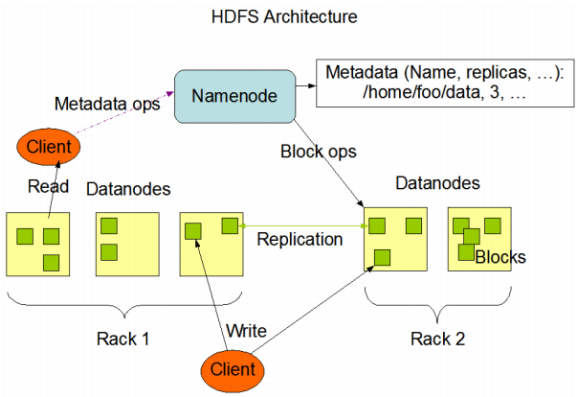

Figure 1: This shows the basic architecture of HDFS. This figure demonstrates the role of the NameNode and DataNodes in HDFS. The HDFS Clients can be seen communicating to the NameNode and then directly to the DataNodes to performs operations such as read and write [2].

4. Fault Tolerance

Fault tolerance is the ability of a system to continue to function correctly and not lose data even after some components of that system have failed [8]. It is difficult to achieve 100% fault tolerance because there are many physical circumstances that just can't be planned for, but the goal of fault tolerance is to plan for all common failures [8]. In managing fault tolerance it is important to eliminate Single Points of Failure (SPOF), which are single elements of the system, that when they fail, they can bring down the whole system [4].

One of the main goals of Hadoop and HDFS is to be highly fault tolerant. Because HDFS can be spread over hundreds or thousands of nodes or machines that can contain cheap, low-cost hardware which makes fault tolerance not a trivial problem [2]. When considering that thousands of computer components and hundreds of network devices such as switches, routers, and power units that are involved in these large distributed systems, it causes

failures to be very frequent. In these systems failures can be daily occurances which makes robust fault tolerance essential for a distributed system such as Hadoop [5].
Hadoop and HDFS center its fault tolerance on data redundancy, which is to replicate data so that if one replica is lost then there are backup copies.

5. Redundancy

Redundancy in HDFS isn't handled using a data protection method such as RAID like other file systems [3]. Instead file content is replicated on multiple DataNodes for reliability. When a client wants to write a file to the HDFS it first contacts the NameNode and then the NameNode nominates three different DataNodes that can be used to replicate the data. The client then writes the data to all of the separate DataNodes which ensures that the client's data is fully replicated throughout the HDFS.
By default the replication factor set by HDFS is three (meaning that all data blocks are replicated three times). However, users can increase the replication factor for specific files especially if they have files that need to be accessed often or contain critical information that would be a disaster to lose. By increasing the replication factor of these critical files the user can ensure that the file has a greater tolerance against faults and having more replicas of a file will also help increase the bandwidth available for reading that file [3].
The location of replicas is also very important for the HDFS reliability and performance. The NameNode uses a rack-aware policy when storing the location of blocks of data on DataNodes [2]. By doing this the NameNode can minimize the bandwidth required to access different blocks of a file, but it also keeps in mind that replicas need to be spread out enough to improve fault tolerance. For example, for a very robust system the NameNode could put replicas on totally separate racks, but this would incur a high communication overhead especially when that file needs to be written by a client because all replicas must be updated during a write procedure. Also, when a block of data is being read, the NameNode will try to select a block that is on the same rack as the node reading the data in order to minimize the communication and bandwidth overhead.
Typically to compromise these two factors in a common three replica file, HDFS will write two of the replicas on different nodes of the same rack and write the other one on a totally separate rack [2].

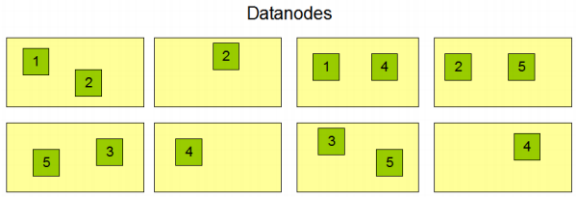

Figure 2: This shows an example of eight DataNodes and how the NameNode can distribute and replicate the blocks of data [2].

In HDFS the blocks of data in DataNodes are well distributed and replicated, but if something happened to the NameNode server then HDFS could fail. HDFS wouldn't be very robust if the NameNode was a Single Point of Failure for the system. In order to minimize the effects of hardware failure of the NameNode, Hadoop introduced the BackupNode [3].
The BackupNode creates periodic checkpoints of the HDFS and also maintains an up to date image copy of the file system. The BackupNode also maintains an exact copy of the namespace from the NameNode. In the event that the NameNode should fail, the BackupNode contains an exact copy of the namespace and image so it can be used to restart the NameNode [3].

6. Job Migration and Location

Due to HDFS's redundancy the same blocks of data are usually found on different nodes or racks. Because of this, it may be more beneficial to perform a read or write on one block of data rather than another based on its location in the system. One rack or node may be under heavy load so it would be better to read the data block not found on that resource. Also, if there is a data block on the rack of the reading node

then there is much less communication overhead to read that block of data rather than one on a different rack.
All of these things have been taken into account in HDFS's implementation. When a client node reads a file it receives a list of data blocks and locations for those blocks. The list of block locations is ordered based on their proximity to the client node [3]. Therefore, it is usually best if the client can access the data blocks listed first because there will usually be less communication overhead which is a significant factor in large distributed systems.
The developers of HDFS also knew that it is usually computationally less expensive to move a computation requested by an application rather than move the data it wants to access [2]. So if a node asks the NameNode to read a particular file and the data blocks for that file happen to be on a different rack or a rack that isn't being used as much then it is better to move the computation closer to the requested node with the data on it. HDFS provides interfaces for applications to perform these necessary moves [2].

7. Weaknesses of HDFS

Hadoops's HDFS has many strengths and does a good job of handling Fault Tolerance, but there are some other aspects of Hadoop and HDFS that could use some improvement that are worth mentioning.
A major problem that Yahoo! recognized is the scalability of the NameNode [3]. Because the NameNode is a single node and not distributed it is subject to physical hardware constraints. For example, because the namespace is kept in memory, if the namespace grows large enough to use most of the node's memory from an expansive number of files in the system then HDFS becomes unresponsive [3]. A possible solution to this is to use more than one NameNode, but that also creates other communication problems [3]. Another way to keep this problem from happening is to try and only store large data files in the HDFS. Large files will reduce the size of the namespace. Obviously, this makes sense because a file system like HDFS is designed for large files.
Another weakness of Hadoop that has been known to affect users [4] is the lack of good high level support for Hadoop. This can happen with open source projects. Enhancing Hadoop's functionality on a system can be difficult without proper support [4].
HDFS is also known to have scheduling delays that keep it from reaching its full potential. This software bottleneck causes some nodes to wait for their new task. This weakness is particularly evident in the HDFS client code [7].

8. Conclusion

Even considering some of the weaknesses of Hadoop's HDFS, it still does a good job of handling fault tolerance. It seems, for HDFS, that fault tolerance is the main focus of Hadoop. In the end, this is very valuable to users because loss of data or a system that crashes can be detrimental to business and can have irreversible consequences. For many this should be a good selling point and reason to use Hadoop.

References

[1] Apache Hadoop. http://hadoop.apache.org

[2] Borthakur, D. (2007) The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf

[3] Shvachko, K., *et al*. (2010) The Hadoop Distributed File System. *IEEE.*

[4] Wang, F. *et al*. (2009) Hadoop High Availability through Metadata Replication. *ACM.*

[5] Bessani, A. *et al.* Making Hadoop MapReduce Byzantine Fault-Tolerant.

http://www.di.fc.ul.pt/~mpc/pubs/bft-mapreduce-fa-dsn10.pdf

[6] Chang, F. *et al.* (2006) BigTable: A Distributed Storage for Structured Data. http://labs.google.com/papers/bigtable.html

[7] Shafer, J. *et al.* (2010) The Hadoop Distributed File System: Balancing Portability and Performance. *IEEE.* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.167.3342&rep=rep1&type=pdf

[8] Selic, B. (2004) Fault tolerance techniques for distributed systems. *IBM.* http://www.ibm.com/developerworks/rational/library/114.html

[9] Zhang, Y. *et al.* (2000) The Impact of Migration on Parallel Job Scheduling for Distributed Systems. *Proceedings of Europar.* *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.1125&rep=rep1&type=pdf*