Hadoop Distributed File System (HDFS)

Thomas Kiencke
Institute of Telematics, University of Lübeck, Germany

***Abstract*—The Internet has become an important part in our life. As a consequence, companies have to deal with millions of data sets. These data sets have to be stored safely and fault tolerant. At the same time hardware should be very cheap. To be faced up with this contrast, Hadoop Distributed File System (HDFS)[1] was developed and fulfils those requirements. The highly available and fault-tolerant framework is designed to run on commodity hardware, where failure is the norm rather than the exception. It provides automatic data replication plus integrity and several interfaces to interact with. HDFS is written in Java and licensed under the open-source Apache v2 license.**

***Keywords*—Hadoop, HDFS, distributed filesystem, RESTful.**

I. Introduction

The requirements on todays file systems are higher than several years ago. Companies like Yahoo, Facebook, or Google need to store large amounts of data sets provided by millions of users and run operations on it. Thats the reason why Apache Hadoop originated. The Hadoop platform provides both, a distributed file system (HDFS) and computational capabilities (Map Reduce). It is highly fault-tolerant and designed to run on low-cost hardware. This paper examines only the distributed file system and not the Map Reduce component. The remainder of the paper is organized as follows. In Section 2, related work of Hadoop is introduced. Section 3 describes the architecture and the functionality of HDFS. Finally, in Section 4 a brief summary and a description of future work are presented.

II. Related Work

There are a couple of different distributed file systems. The two most auspicious ones are HDFS and Googles proprietary version Google File System (GFS). Both file systems have a lot in common, as HDFS was derived from Googles GFS papers [2], but astonishingly released before Googles file system.

Besides the distributed file system Hadoop has numerous extension possibilities [3]:

- *Avro*: A data serialization system.
- *Cassandra*: A scalable multi-master database with no single points of failure.
- *Chukwa*: A data collection system for managing large distributed systems.
- *HBase*: A scalable, distributed database that supports structured data storage for large tables.
- *Hive*: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- *Mahout*: A Scalable machine learning and data mining library.
- *Pig*: A high-level data-flow language and execution framework for parallel computation.
- *ZooKeeper*: A high-performance coordination service for distributed applications.

III. The Design of HDFS

In this section the architecture and the functionality of HDFS are described.

A. What is HDFS?

HDFS is a distributed file system, written in Java, which allows storing very large files and running on clusters on commodity hardware. Large files mean in this context files that are gigabytes or terabytes in size. The newest Hadoop versions are capable of storing petabytes of data. A big advantage compared to other big data approaches (such as mainframes) is that Hadoop does not require expensive, highly reliable hardware to run on. It is designed to run on clusters of commodity hardware. HDFS carries on working without a noticeable interruption to the user if individual nodes of the cluster fail [4]. HDFS is not comparable to a normal file system.The requirements of the POSIX (Portable Operating System Interface) standard, which describes the interface between application software and the operating system [5], cannot entirely be fulfilled. HDFS relaxes a few requirements to enable streaming access to the file system data. In the course of the paper differences between the hadoop implementation and the POSIX standard will be shown. Nevertheless, HDFS provides various capabilities to interact with the file system, which will be described in more detail later.

B. What is HDFS not for?

The applications that use HDFS are not general purpose applications. The file system is more designed for batch processing rather than for an interactive use by users. Thus emphasis is on high throughput instead of low latency of data access [6]. Applications that need to work in tens of milliseconds range will not work well with HDFS. In that case a better choice would be HBase, which is optimized for real time read/write access for Big Data [7]. Another point where HDFS is not a good fit is the work with a big amount of small data files. The reason for that is the management overhead of the data(-blocks), which will be explained later. Furthermore, the file system does not support multiple writers at the same time and modifications at arbitrary offsets in the file, hence writes are always made at the end of the file. Instead it is optimized for multiple read operations and high throughput [4].

C. The concept of Blocks

A normal file system is separated into several pieces called blocks, which are the smallest units that can be read or written. Normally the default size is a few kilobytes. HDFS also has

the concept of blocks, but of a much larger size, 64 MB by default. The reason for that is to minimize the costs of seeks for finding the start of the block. With the abstraction of blocks it is possible to create files that are larger than any single disk in the network.

D. Namenodes and Datanodes

HDFS implements the master-worker-pattern, which means there exists one namenode (the master) and several of datanodes (workers). The namenode is responsible for the management of the file system namespace. It maintains the file system tree and stores all metadata of the files and directories. In addition the namenodes know where every part (block) of the distributed file is stored. The datanodes store and retrieve all blocks when they are told by the namenode and serve reading and writing requests from the system clients. Furthermore, they report back periodically a list of blocks which they are storing [4]. As the namenode is the only entry point for the client

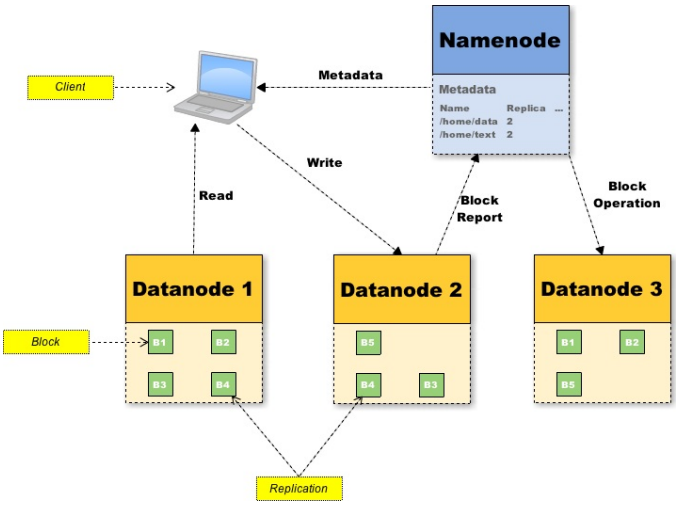

Fig. 1. HDFS architecture

to find out which datanode is responsible for the data, it is a single point of failure. If the namenode crashes, the file system cannot be used. To solve this problem Hadoop provides two mechanisms [4]:

- Every operation is written to a transaction log called edit log. These writes are synchronous and atomic. In practice data are often stored to the local disk and in addition on a remote NFS mount. On startup the machine reads the edit log from disk and applies all missing transactions.
- Another way is to run a secondary namenode, which does not act like a real namenode. Its main purpose is to merge periodically the namespace image with the edit log to prevent the edit log from becoming too large. This secondary node usually runs on a physical separated machine. However, the state of the node lags behind that of the primary node, which means that if the primary one crashes and loses all of its data a data loss is almost guaranteed.

In any case, if a namenode machine fails, a manual intervention is necessary. Currently, automatic failover of the namenode software to another machine is not supported (status: December 2012) [8].

E. Data Replication

For providing fault tolerance and availability each block is replicated to other machines. The default replication factor is three. But it is also possible to change this default value or to specify it individually for different files. If a node is temporarily not available, the request will be redirected to a replication node. In case of corruption or machine failure the block will be replicated again to bring the replication factor back to the normal level. All replications are managed by the namenode. Each datanode periodically sends a heartbeat and a block report (contains a list of the all blocks on that datanode) to the namenode. So the namenode can decide if it is necessary to create a new replication or if everything works properly [4]. The replications are distributed in a special way called rack awareness to improve data reliability, availability and network bandwidth utilization. Large HDFS instances run on thousands of computers that are separated into many racks. These racks communicate through switches. Since the communication between two nodes in the same rack is faster than the communication between two nodes in different racks, in this case HDFS prefers the two nodes in the same rack. In addition, it is also possible to use the full bandwidth of all racks while reading multiple blocks. On the other hand, the software takes care of storing at least one replication block in a different rack, which prevents losing all data if a single rack fails [9].

F. Staging and Replication Pipelining

When a client writes a new file to the distributed file system, it is first written to the local disk. This cache mechanism is called staging and is needed to improve the network throughput [10]. That implementation is different to the POSIX standard. The file is already created, but other users cannot read the file until the first part reaches the full block size and has been sent to a datanode. After the file has reached the full block size, the client retrieves a list of datanodes from the namenode. The client can choose one of them, most likely the nearest one, and then send the block to this datanode. Now this node is responsible for sending the block to the second node, and the second node for sending it to the next node and so on. This mechanism is called replication pipelining [11] and it is needed to bring up the replication factor to a given level. As the client is only responsible for sending the block to one datanode, this mechanism relieves the resources of the client.

G. Persistence of the Data

The whole file system namespace is stored in a local file called fsimage by the namenode. The datanode stores all blocks in its local file system. These blocks are separated into different directories, because not every local file system is capable to handle a huge amount of files in one directory

efficiently. The datanode has no knowledge about the HDFS files. On startup the datanodes scan all directories and generate a list with all files (the block report) [12].

H. Data Integrity

Every time a new HDFS file is created, the namenode stores the checksums of every block. When a client retrieves this HDFS file, the checksum can verify the integrity. If the checksums don't match, the file will be retrieved from another datanode.

I. Authentication

With the default configuration, Hadoop doesnt do any authentication of users. In Hadoop the identity of a client process is just whatever the host operating system says it is. For Unix-like systems it is equivalent of 'whoami' [13]. But, Hadoop has also the ability to require authentication, using Kerberos principals. With the Kerberos protocol it is possible to ensure that when someone makes a request, they really are who they say they are [10].

J. Authorization

HDFS implements a permission model that resembles the POSIX model. Each file and directory is associated with an owner and a group. The file or directory has separate permissions for the user that is the owner, for other users that are members of the group and for all other users. For files, the r permission is required to read the file, the w permission is required to write or append to the file and the x permission is ignored since it is not possible to execute a file on HDFS. For directories, the r permission is required to list the contents of the directory, the w permission is required to create or delete files or directories, and the x permission is required to access a child of the directory. When a file or directory is created, its owner is the current user and its group is the group of the parent directory [14].

K. Configuration

The configuration files are placed in the conf directory. There are two important files for starting up HDFS. The first one is the core-site.xml which is the generic configuration file for Hadoop and the second one is the hdfs-site.xml file where HDFS specific configurations are stored. These two files are empty by default. If no new properties are added, default values are used. The default configuration for the core and HDFS component can be found in the documentation ([15] and [16]). To modify default values it is necessary to add the corresponding value to the configuration file. For example this snippet would change the replication factor to 1:

```
<!-- In: conf/hdfs-site.xml -->
<property>
  <name>dfs.replication</name>
  <value>1</value>
</property>
```

L. Accessibility

HDFS provides several ways to access the file system.

1) Java Interface: The native interface is the Java API [17]. First of all it is important to import the necessary libraries. The easiest way is to include the hadoop-core package as a maven dependency:

```
<dependency>
        <groupId>org.apache.hadoop</
           groupId>
        <artifactId>hadoop-core</
           artifactId>
<version>1.1.1</version>
</dependency>
```

Before any request can be performed, the Hadoop configuration has to be specified:

```
Configuration conf = new Configuration();
conf.addResource(new Path("/app/hadoop/
   conf/core-site.xml"));
conf.addResource(new Path("/app/hadoop/
   conf/hdfs-site.xml"));
```

In this case, these are the original configuration files, but it is also possible to reference a copy in the classpath.

Afterwards, it is possible to create a reference to the distributed file system:

```
FileSystem fileSystem = FileSystem.get(
   conf);
```

Another way is to create an empty configuration object and specify the whole HDFS URI:

```
Configuration conf = new Configuration();
String uri = "hdfs://localhost/";
FileSystem fs = FileSystem.get(URI.create(
   uri), conf);
```

Below the basic operations are illustrated:
create a new directory:

```
fileSystem.mkdirs(path);
```

delete a file:

```
fileSystem.delete(path, true);
```

read a file:

```
FSDataInputStream input = fileSystem.open(
   path);
StringWriter output = new StringWriter();
IOUtils.copy(input, output, "UTF-8");
String result = output.toString();
```

add a file:

```
String content =   my  content ;
FSDataOutputStream output = fileSystem.
   create(path);
output.write(content.getBytes(Charset.
   forName("UTF-8")));
```

Other operations can be found in the file system API [18]. See Appendix A for a complete working example.

2) Command Line: In addition there exists a command line interface that is called fs shell. With this shell it is possible to interact with the file system. The syntax is similar to Unix shells like bash or csh. To call the fs shell it is necessary to execute the Hadoop binary with the parameter dfs. Here are example commands:

a new directory:

```
hadoop dfs -mkdir /path/to/directory
```

recursive remove a directory:

```
hadoop dfs -rmr /path/to/directory
```

read a file:

```
hadoop dfs -cat /dir/file.txt
```

copy a file from a local disk to the distributed file system:

```
hadoop dfs -put localfile /dir/hadoopfile
```

copy a HDFS file to the local disk:

```
hadoop dfs -get /user/hadoop/file
localfile
```

There are a few other commands which can be found on the project page [19].

3) Interface for other programming languages: As HDFS is written in Java it is difficult to interact with the file system from other programming languages. Therefore HDFS provides a Thrift interface called thriftfs. Thrift is an interface definition language that is used to define and create services that are both consumable by and serviceable by numerous languages [20]. This interface makes it easy for any language that has a Thrift binding to interact with the file system. To use the Thrift API, it is necessary to start a Java server that exposes the services and acts as a proxy to the file system [21].

4) Mounting HDFS: It is also possible to integrate the distributed file system as a Unix file system. There are a couple of projects for example fuse-dfs, fuse-j-hdfs or hdfs-fuse provide this functionality. All of them are based on the file system in userspace-project FUSE [22]. Once mounted, the user can operate on an instance of HDFS using standard UNIX utilities like ls, cd, cp, mkdir, find, grep, or standard POSIX library functions such as open, write or read from C, C++, Python, Ruby, Perl, Java, bash, etc. [23].

5) REST Interface: Another way to open the door for other languages expecting Java is to access HDFS using standard RESTful Web services. The WebHDFS REST API supports the complete file system interface for HDFS. With this interface even thin clients can access via web services, without installing the whole Hadoop framework. In addition WebHDFS provides authentication support via Kerberos and a JSON format for status data like file status, operation status or error messages. With the interface it is very easy to interact with the file system. Here are some example commands using the command-line tool curl to create and send the HTTP messages:

Create and Write to a File [24]:

Step 1: Submit a HTTP PUT request without automatically following redirects and without sending the file data

```
curl -i -X PUT "http://<HOST>:<PORT>/
    webhdfs/v1/<PATH>?op=CREATE[&overwrite
    =<true|false>[&blocksize=<LONG>][&
    replication=<SHORT>]ermission=<OCTAL
    >][&buffersize=<INT>]"
```

The request is redirected to a datanode where the file data is to be written:

```
HTTP/1.1 307
TEMPORARY_REDIRECT
Location: http://<DATANODE>:<PORT>/webhdfs
    /v1/<PATH>?op=CREATE...
Content-Length: 0
```

Step 2: Submit another HTTP PUT request using the URL in the Location header with the file data to be written:

```
curl -i -X PUT -T <LOCAL_FILE>"http://<
    DATANODE>:<PORT>/webhdfs/v1/<PATH>?op=
    CREATE..."
```

The client receives a 201 Created response with zero content length and the WebHDFS URI of the file in the Location header:

```
HTTP/1.1 201 Created
Location: webhdfs://<HOST>:<PORT>/<PATH>
Content-Length: 0
```

List a Directory [25]:

```
curl -i  "http://<HOST>:<PORT>/webhdfs/v1
    /<PATH>?op=LISTSTATUS"
```

The client receives a response with a FileStatuses JSON object:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 427

{
  "FileStatuses":
  {
    "FileStatus":
    [
      {
        "accessTime"      : 1320171722771,
        "blockSize"       : 33554432,
        "group"           : "supergroup",
        "length"          : 24930,
        "modificationTime": 1320171722771,
        "owner"           : "webuser",
        "pathSuffix"      : "a.patch",
        "permission"      : "644",
        "replication"     : 1,
        "type"            : "FILE"
      },
      ...
    ]
  }
}
```

The whole API description can be found on the project site [26]

6) Web Interface: The namenode and datanode each run an internal web server in order to display the basic information about the current status. It is also possible to run file browsing operations on it. By default the front page is at http://namenode-name:50070/ [27].

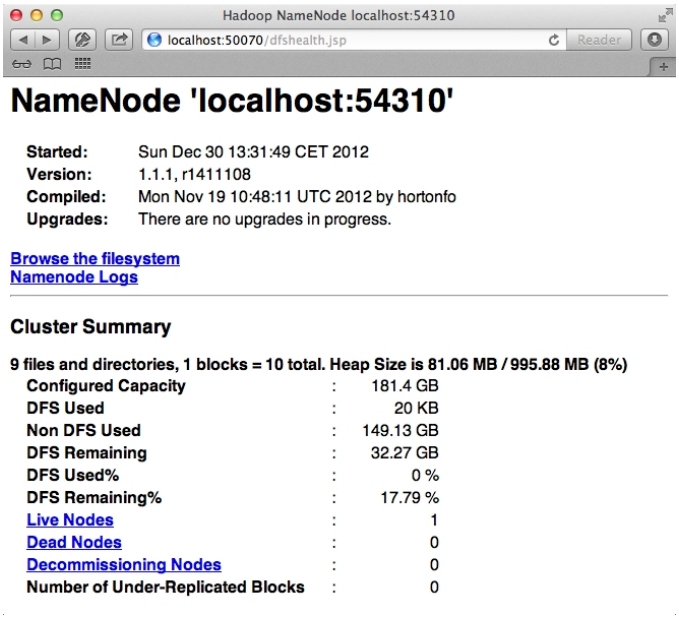

Fig. 2. HDFS Web Interface

IV. Summary and Future Work

As we have seen, HDFS is a powerful and versatile distributed file system. It is easy to configure and operate with through its many interfaces. As Hadoop/HDFS is built and used by a global community, it is easy to start with. The documentation from Hadoop itself and other tutorials/articles all over the internet are very good. Nevertheless, there are some points where Hadoop should improve in the future. Since Hadoop wants to provide a highly available platform, automatic failover of the namenode software to another machine should be supported. Furthermore, HDFS should integrate a better authentication/authorization model. For example the current version doesnt check if a user really exists while changing the file/directory owner.

Appendix A

```
import java.io.IOException;
import java.io.StringWriter;
import java.nio.charset.Charset;
import org.apache.commons.io.IOUtils;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.conf.Configuration;

public class HDFS {

        public static void main(String[] args)
            throws IOException {
                // set up configuration
                Configuration conf = new
                    Configuration();
                conf.addResource(new Path("/app/
                    hadoop/conf/core-site.xml"));
                conf.addResource(new Path("/app/
                    hadoop/conf/hdfs-site.xml"));

                createDir(conf, "/home");

                createDir(conf, "/home/user");

                addFile(conf, "/home/user/test.txt",
                     "no␣content␣:)");

                System.out.println(readFile(conf, "/
                    home/user/test.txt"));

                deleteFile(conf, "/home/user/test.
                    txt");

        }

        private static void createDir(Configuration
            conf, String dirpath)
                        throws IOException {
                FileSystem fileSystem = FileSystem.
                    get(conf);
                Path path = new Path(dirpath);
                if (fileSystem.exists(path)) {
                        System.out.println("
                            Directory␣already␣␣
                            exists");
                        return;
                }
                fileSystem.mkdirs(path);
                fileSystem.close();
        }

        private static void deleteFile(Configuration
             conf, String filepath)
                        throws IOException {
                FileSystem fileSystem = FileSystem.
                    get(conf);

                Path path = new Path(filepath);
                if (!fileSystem.exists(path)) {
                        System.out.println("File␣
                            does␣not␣exists");
                        return;
                }

                fileSystem.delete(path, true);
                fileSystem.close();
        }

        private static void addFile(Configuration
            conf, String filepath,
                        String content) throws
                            IOException {
                FileSystem fileSystem = FileSystem.
                    get(conf);

                Path path = new Path(filepath);
                if (fileSystem.exists(path)) {
                        System.out.println("File␣
                            already␣exists");
                        return;
                }
```

```
                FSDataOutputStream output =
                    fileSystem.create(path);
                output.write(content.getBytes(
                    Charset.forName("UTF-8")));

                output.close();
                fileSystem.close();
        }

        public static String readFile(Configuration
            conf, String filepath)
                    throws IOException {
                FileSystem fileSystem = FileSystem.
                    get(conf);

                Path path = new Path(filepath);
                if (!fileSystem.exists(path)) {
                        System.out.println("File␣
                            does␣not␣exists");
                        return null;
                }

                FSDataInputStream input = fileSystem
                    .open(path);
                StringWriter output = new
                    StringWriter();
                IOUtils.copy(input, output, "UTF-8")
                    ;
                String result = output.toString();
                input.close();
                output.close();
                fileSystem.close();
                return result;
        }
}
```

REFERENCES

[1] [Online]. Available: http://hadoop.apache.org/

[2] [Online]. Available: http://users.ece.gatech.edu/~dblough/6102/presentations/gfs-sosp2003.pdf

[3] [Online]. Available: http://hadoop.apache.org/#What+Is+Apache+Hadoop%3F

[4] T. White, *Hadoop: The Definitive Guide*, ser. Definitive Guide Series. O'Reilly Media, Incorporated, 2009.

[5] [Online]. Available: http://standards.ieee.org/develop/wg/POSIX.html

[6] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/hdfs_design.html#Streaming+Data+Access

[7] [Online]. Available: http://hbase.apache.org/

[8] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/hdfs_design.html#Metadata+Disk+Failure

[9] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/hdfs_design.html#Data+Replication

[10] [Online]. Available: http://blog.cloudera.com/blog/2012/03/authorization-and-authentication-in-hadoop/

[11] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/hdfs_design.html#Replication+Pipelining

[12] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/hdfs_design.html#The+Persistence+of+File+System+Metadata

[13] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/hdfs_permissions_guide.html#User+Identity

[14] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/hdfs_permissions_guide.html#Overview

[15] [Online]. Available: http://hadoop.apache.org/docs/r1.0.0/core-default.html

[16] [Online]. Available: http://hadoop.apache.org/docs/r1.0.0/hdfs-default.html

[17] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/api/

[18] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/api/org/apache/hadoop/fs/FileSystem.html

[19] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/file_system_shell.html

[20] [Online]. Available: http://jnb.ociweb.com/jnb/jnbJun2009.html

[21] [Online]. Available: http://wiki.apache.org/hadoop/HDFS-APIs

[22] [Online]. Available: http://fuse.sourceforge.net/

[23] [Online]. Available: http://wiki.apache.org/hadoop/MountableHDFS

[24] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/webhdfs.html#CREATE

[25] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/webhdfs.html#LISTSTATUS

[26] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/webhdfs.html

[27] [Online]. Available: http://hadoop.apache.org/docs/r1.1.1/hdfs_user_guide.html#Web+Interface