EECS4314 Assignment 2: Concrete Architecture of Hadoop Distributed File System

Randy Agyapong
David Iliaguiev
Hashim Al-Helli
Zhongran (Julian) Deng
Sied Hoa (Heny) Tjin

November 2, 2016

Contents

1 Abstract

This report provides an overview of the concrete architecture of Hadoop Distributed File System (HDFS) from Apache Hadoop. We will first discuss the general architecture of Apache Hadoop and HDFS, as well as the conceptual architecture of HDFS. We will discuss the methods used to derive the concrete architecture directly from the source code. We will then discuss each subsystem extracted in detail, and how they relate to the conceptual architecture. These subsystems are Name Node, Data Node, Protocol, Common, Client, Block Management and Balancer. We'll produce a finalized concrete architecture and a modified conceptual architecture of HDFS. We'll describe a use case that utilizes HDFS. Finally, we will discuss lessons learned when creating this report.

2 Introduction and Overview

Data can be classified as Big Data when it becomes difficult to store, search, or even manage data using traditional management tools. Such Big Data may reach sizes in the petabytes. Even the Internet produces petabytes of data daily, whether it would be info generated by the millions of users browsing, or automatically through systems and programs. Managing these large scales of data in a reliant and efficient manner ultimately led to the creation and development of Apache Hadoop.

Apache Hadoop is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. Apache Hadoop is scalable and reliable due to the replication of data node to accommodate failure. Its high level conceptual architecture can be seen in Figure (2). Although YARN and Mapreduce are major components in Apache Hadoop, we will delve into HDFS.

HDFS, whose full name is Hadoop Distributed File System, is an open-source, distributed, scalable and portable file system written in Java for the Hadoop framework. HDFS is designed for commodity hardware to store and process big data across multiple machines to achieve parallel computing. It provides an ideal pattern to store large amounts of data overcoming the limit of traditional data storage pattern [1]. It provides a foundational file system structure for MapReduce algorithm. Basic concepts of HDFS include HDFS cluster, name node, data node, data block, data block replication, secondary name node, and backup node. Their purpose will be explained in the subsystem chapters on page 6. Fundamentally, the conceptual architecture of HDFS can be seen in Figure (1). This architecture was found on the official Hadoop 2.7.3 documentation.

Throughout the report, we will go into detail how we extracted and derived the concrete architecture using various methods. LSEdit, Understand, Notepad++, and MicroExcel provided means to extract and fully analyze each subcomponent in HDFS. Each subsystem will be fully explained in detail as well as their connection to the concrete architecture.

3 Description of Concrete Architecture

HDFS Principle and Architecture Design

HDFS is designed based on the following principle: [4]

- **Hardware Failure**: Since the system utilizes numerous commodities to run as DataNode, hardware failure is inevitable and it should be handled by the system.
- **Streaming Data Access**: To achieve high throughput, rather than low latency, access to data and data transfer is done by streaming.
- **Large Data Sets**: HDFS is designed to support large data sets by providing high aggregate data bandwidth and scale to hundred of nodes in a single cluster.
- **Simple Coherency Model**: HDFS is a write-once-read-many access model for files. This is implemented to achieve simplified data coherency issues and high throughput data access. With write-once-read-many model, it eliminates the constant need to synchronize data between Data Nodes, which will be impractical in distributed file system platform.

HDFS stores file system metadata and application data separately, with host server of NameNode storing the metadata, and other nodes of DataNodes storing the client's data. All nodes are connected and communicate with each other using TCP-based protocols [3]. HDFS implements replication on multiple DataNodes for fault-tolerance. Replication also serves another purpose: there will be more than one copy of data, and hence there will be more opportunities for locating computation near the needed data [3].

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. A datanode stores and transfers data, and it constantly sends its status report to NameNode.

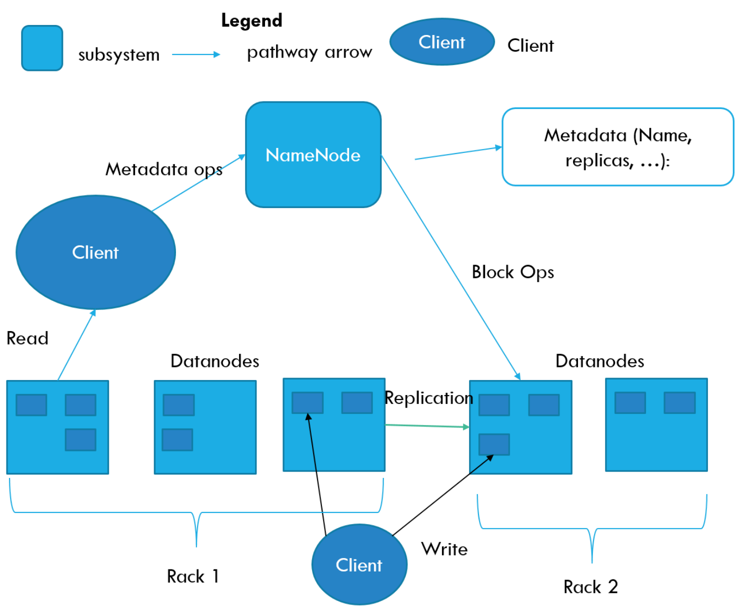

Figure 1. Conceptual Architecture of HDFS

Extraction Methodology
We delved into the source code of Apache Hadoop version 2.7.3 to determine the dependency relationships in HDFS. The size of the implementation is very large (over 6000 java classes). It is not feasible to do manual examination of going through every class. Hence, we used automated tools to extract relations from the source code, and use the extracted relations to draw out the concrete system architecture.

We used GROK to analyze the source code of HDFS and make modifications to our extractions in iterations until we found a satisfiable concrete architecture. We first used Understand to extract dependency relations, such as “class A depends on class B”. The class dependencies report generated by Understand is a set of relations between classes. The number of relations are still too large for manual examination.

Our goal is to examine relations between main sub-systems. Hence, we use relations between clustering of files to determine relations between subsystems. We further use LSEdit, Notepad++ and MicroExcel to determine relations between the packages.

Based on the visualized tool of LSEdit, we drew out the hierarchical decomposition of the system, and determined the main package that contain the subsystems of HDFS. This decomposition is illustrated in Figure (3).

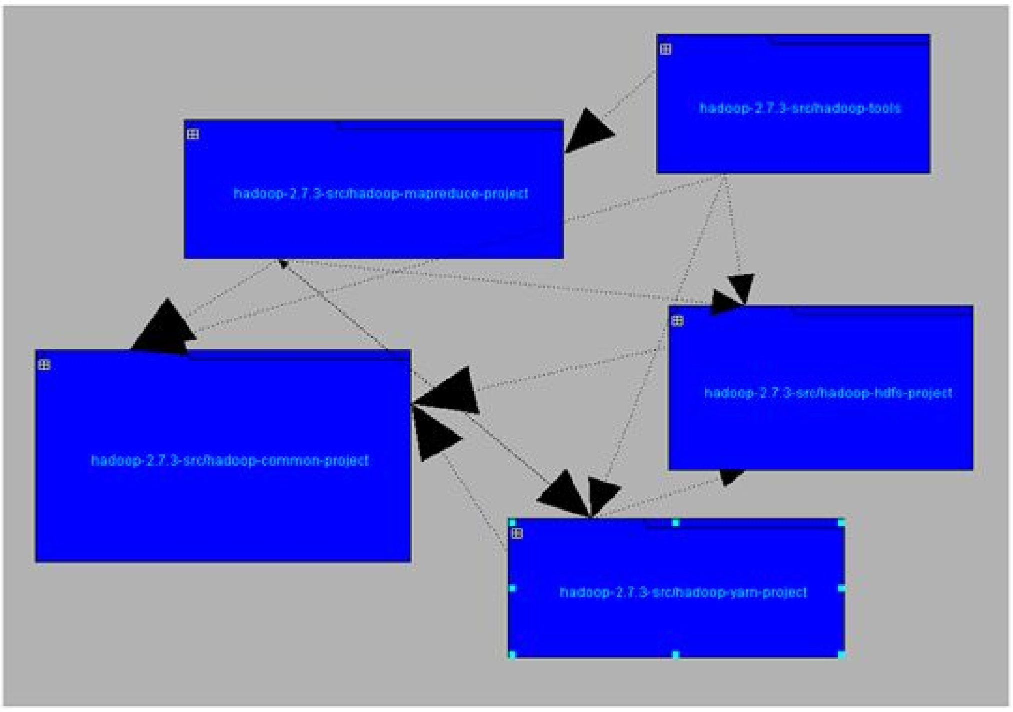
hadoop-2.7.3-src/hadoop-tools
hadoop-2.7.3-src/hadoop-mapreduce-project
hadoop-2.7.3-src/hadoop-hdfs-project
hadoop-2.7.3-src/hadoop-common-project
hadoop-2.7.3-src/hadoop-yarn-project

Figure 2: Apache Hadoop System Conceptual Architect at highest Level, extracted using LSEdit

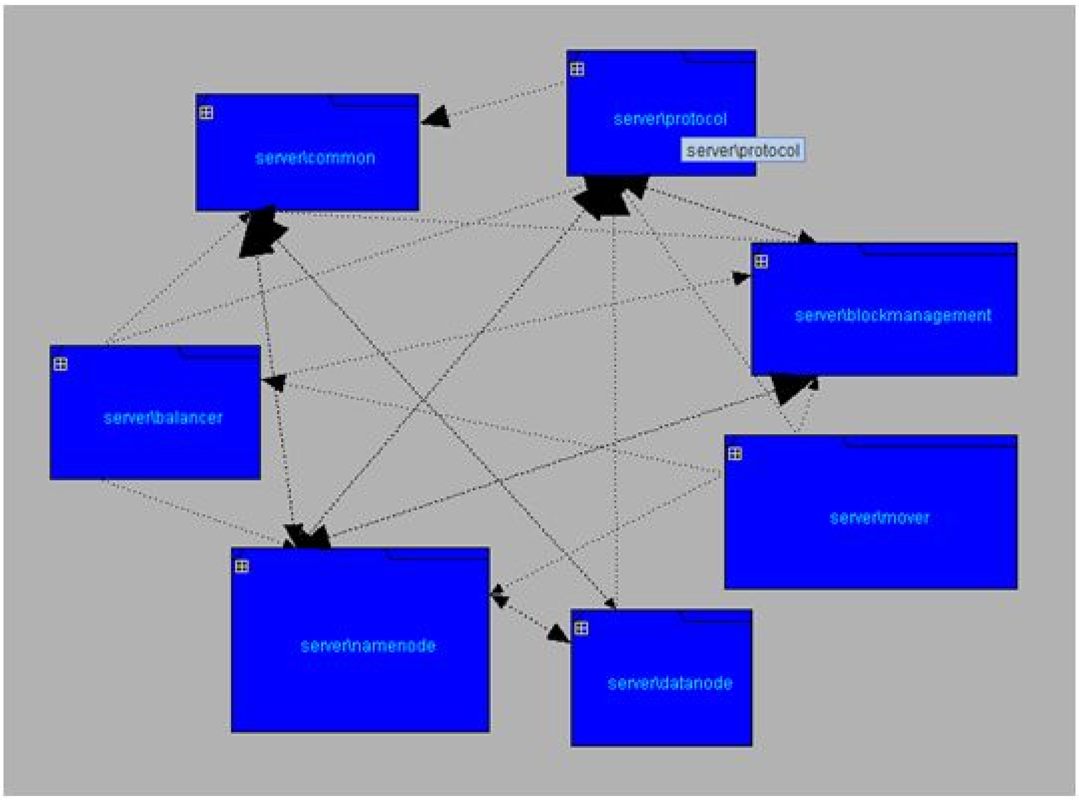

Figure 3: HDFS Original Concrete Architect, extracted using LSEdit (extracted from directory: ...\hdfs\server)

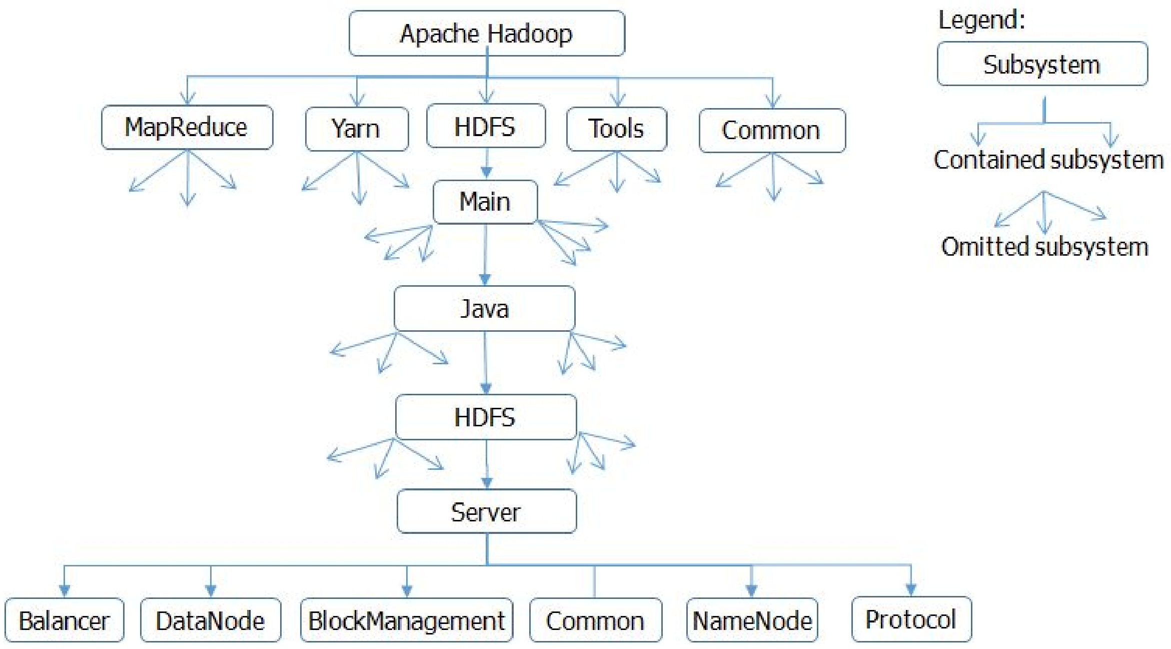

Figure 4: Hierarchical Decomposition - Partial Subsystem Hierarchy

We also utilized the powerful text processing functionality of Notepad++ along with its plug-in TextFX to process the original ls.ta file and extract the dependency relations that we need. Starting from the original records with each keyword filtering, we filtered out the records that were not needed such as MapReduce or test tool for hdfs. After we derived all records under HDFS/server (where most HDFS function codes are stored), we searched the keyword of the subsystems pairwisely. Thus we derive the dependency relation pairwisely and we could start our investigation.

4 Subsystems of HDFS

4.1 NameNode

NameNode functions as control center under the HDFS conceptual architecture. It stores metadata information of the file system. It manages DataNodes in which NameNode knows of where a certain data file block is stored. It holds fsimage and editlog which are used to capture the status of namenode, functioning like snapshots. It interacts with almost all other subsystems in the HDFS system.

Concrete internal architecture:

The subsystem of NameNode is clustering several modules of different functionalities together. Secondary node is considered as subsystem distinct from NameNode [2], yet in our concrete architecture diagram, it is included in NameNode system. This situation also applies to checkpointer and back-up node. An HA (High Availability) module is also included to support the implementation of standby nameNode. Two persistent files, Fsimage and editlog, are also included to facilitate the checkpoint functionality.

Before the architecture repairing:

In the diagram derived from LSEdit (figure 3), NameNode communicates with almost every other subsystems, which also coincide with our intuition as well as the conceptual architecture. NameNode is involved in many operations inside HDFS system as it stores metadata information of the file system and knows of the actual location of a certain data block.

Repairing the concrete architecture

1. Remove dependency relation between DataNode and NameNode

In the original conceptual architecture in figure (1), NameNode manages a bunch of DataNodes. Intuitively, it is thought that NameNode would have a very strong coupling relation with datanode as communication between namenode and datanode should be frequent. However, we will find that the coupling relation is not strong as shown in the LSEdit generated diagram (figure 3) where arrows are relatively small. After investigation into dependency records, we have 8 weak dependencies which should be removed. In fact, NameNode achieves communication with datanode mostly via Protocol.

2. Remove dependency relation between balancer and namenode.

Balancer is a tool that balances disk space usage on an HDFS cluster when some DataNodes become full or when new empty nodes join the cluster. As we can see from the LSEdit generated diagram (figure 3), the dependence relationship between balancer and namenode is weak. In our opinion, this relation should be removed because the exception class in Balancer does contain any routines that could be run to change status of any other models.

Interaction with other system

Strong dependence relation is manifested between NameNode and BlockManagement. This is because NameNode manages DataNodes/data blocks via BlockManagement. A strong dependence relation is also manifested with Common and Protocol because these two subsystem are utilized by all other subsystem.

4.2 DataNode

DataNode functions to store the data blocks under the management of NameNode. It communicates with the NameNode through the Protocol subsystem as it can not contact it directly. DataNodes also contact the client through the protoolPB package.

When the concrete architecture of HDFS was initially extracted, DataNode depended on a lot of subsystems, and vice versa. We realized the only method that DataNode communicates with other systems, specifically NameNode, is through the Protocol subsystem. The dependency arrows between the two were removed as they do not make direct contact with each other.

Comparison between Conceptual Architecture and Concrete Architecture:

This revision also contradicts the conceptual architecture of HDFS. In actuality, there is no bidirectional dependency between NameNode and DataNode. We revised this in figure 6.

4.3 Protocol

The Server.Protocol subsystem consists of classes relating to protocols. These are essentially communication methods in which two subsystems can interact with each other. Protocols are a gateway for communication between two subsystems, and without the existence of these protocol classes, these subsystems would be unable to interact with each other.
The purpose in creating a Protocol subsystem is for reusability. If one were to make changes to a method that the NameNode contacts the DataNode with, the DataNode class does not not need to be changed. Only the Protocol class changes. This also decouples the subsystems from Protocol, easing the way for developers to add new functionality without fully changing the code of other subsystems.

A simple pipeline processing design pattern is used. If we were to consider the DataNode sending information about its state to the DataNode, it can be viewed as a process of events with three entities. The first entity, DataNode, sends its request wrapped in a Protocol. Specifically, blockReport() would be issued in the protocol interface for the DataNode, and it returns a NamenodeCommand object. The Protocol would be the second entity. The DataNode provides all the information it needs in that Protocol class, and the third entity NameNode can finally access that request through the NamenodeCommand object and act accordingly. The request is pipelined linearly from DataNode to Protocol, and from Protocol to NameNode. The reverse is also true.

Before the architecture repair: It was discovered through LSEdit that most, if not all the main subsystems in HDFS depend entirely on Protocol. Looking back on figure (3), each subsystem has a dependency arrow directed at the Protocol subsystem. This makes sense considering the only way each subsystem can communicate with each other with through the Protocol subsystem.

After the architecture repair: After removing the weaker dependency relationships between each subsystem, it was ultimately decided that the dependencies Protocol has should remain. The main functions called in a subsystem, for example Balancer, call methods in Protocol.

Comparison between Conceptual Architecture and Concrete Architecture:

The conceptual architecture had only three main subsystems in figure (1): DataNode, NameNode, and the Client. Each subsystem had bidirectional arrows pointing to each other, so each system depended on another. There was no place for the Protocol subsystem. In the concrete architecture we chose to include Protocol since it was the only way each subsystem can directly communicate with each other, and it also showed how decoupled the subsystems are. In figure (6), the modified conceptual architecture now contains Protocol.

4.4 Common

The common subsystem is an important component for HDFS. It contains utilities that are used by the major subsystems such as NameNode, DataNode and Balancer subsystems. The classes in the common subsystem cover general logic that is shared amongst the subsystems. Common covers system wide constants, data storage information, abstract data types and their exceptions, and web script changes for http requests from the server.

The class HdfsServerConstants is one of most important classes as it contains constants for the server such as start-up options, the setting and getting of cluster IDs, forcing values such as formats of storage data, defining NameNode roles, different node type constants and other system wide constants that can be used. This abstracts common variables so that other subsystems' classes are not cluttered with multiple versions of the same constant, relieving messy source code also known as spaghetti code. The subsystems that depend on this class are NameNode, DataNode, Protocol, BlockManagement, Balancer as well as several classes under the main HDFS system.

The Util class deals with Uniform Resource Identifiers (URI). URI is a string of characters that is used to identify resources. This class takes strings representing files and paths as parameters to transfer to URIs. The NameNode subsystem depends on this class to find directory names. Looking at how the NameNode creates these directories, we can see a design pattern being used. The NameNode uses the Singleton design pattern which makes sure that there is only ever one instance of an object, this is important as there should only be one instance of a particular path otherwise there could be name clashes as well as issues accessing and modifying those directories.

The StorageInfo class is a deferred abstract data type that contains information about node storage such as the layout version, the namespace ID, the cluster ID and the creation time. The Storage class inherits the former and implements the methods for the class. The NameNode has a class NNstorage and DataNode has a class datastorage that inherit the Storage class and thus depend on the common subsystem to maintain information differently for each subsystem. These two subsystems that depend on this class also use the Iterator design pattern, which provides an abstraction to iterate over a list of objects, to go through lists of storage information to do some processing with it. The NameNode and DataNode subsystems also depend on other classes within common that deals with the storage abstract data type such as StorageErrorReporter that gives errors for information trying to be set into the data type as well as InconsistenFSStateException and IncorrectVersionException classes to deal with exceptions resulting from the errors mentioned.

4.5 Client

Client component is a code library that exports the HDFS file system interface. It provides services similar to conventional file systems, which is to read, write, create, delete, rename, open and close files (or directories).

When reading, Client communicates with NameNode for the list of DataNodes that store blocks of a particular file. It then communicates with DataNodes directly for the data transfer. When writing, Client communicates with NameNode for the chosen list of DataNodes to store replicas of the first block of file, and the subsequent datablocks via pipelining: After transfer of first block,a new pipeline is organized and Client sends further bytes of the file.

Architecture Repair

Based on our conceptual architecture, Client is the starting point of many processes. However, we could not decipher this just by looking at the package within 'hdfs.server.client'. After further investigation, the main client function is located one directory up at 'hdfs.DFSClient.java'.

Furthermore, we discovered that the client component also encompass objects to facilitate the staging process prior to flushing blocks of data to DataNodes. Staging is the process whereby transferred data is cached locally at the client's machine until it reaches the size of a block of data. A couple of the main helper components of this process are the classes that create the OutputStream and InputStream objects. Rationale for this process is to facilitate streaming of data instead of transferring using RPC. If remote data transfer is used, the performance will be greatly impacted as it can affect the network speed and can cause congestion in the network.

Client communicates to both DataNode and NameNode via protocol. Rationale for this process is to decouple Client from both major components, so each subsystem's code can be altered without affecting each other.

We also found out that Client initiates the pipeline process. The pipeline process is the process of transferring a sequence of blocks of data of the transferred file.

Another discovery is that Client is the one who notified NameNode for completion of data transfer or if there it gets faulty data. Originally, we envisioned it will be DataNodes who has this responsibility. The changes were altered in figure (5) and figure (6).

4.6 Block Management

Blockmanagement is not in the original conceptual architecture of HDFS. We are a little thrown back upon discovering its existence within the HDFS. After looking through the dependencies and briefly viewing some of the implementation, we found out that Blockmanagement is a major component that facilitate DataNodes in managing blocks. It is the main façade component between DataNode and NameNode for periodically sending of heartbeat and blockreport. It also the main component that facilitates the block replication process, by realizing the 'rack-awareness' property by communicating with Namenode to determine the next location for the block replication.

4.7 Balancer

Balancer is another component that we did not anticipated to come across while extracting the concrete architecture of HDFS. After looking through some resources, we found out Balancer is an added tool of the system that balanced disk usage on HDFS cluster. It analyzes block placement and balances data across the DataNodes. Blocks of data are moved until the cluster is deemed balanced, where the storage utilization of every DataNode differs relatively from the utilization of the whole cluster by no more than a given threshold percentage, usually 10% by default [3].

Rationale of rebalancing: As the distribution of data does not consider utilization of DataNode, data blocks might not always be placed uniformly across data nodes. Imbalances is further impacted when there is new nodes added into the cluster, as the new node can become bottleneck because all new blocks will be allocated and read from that datanode.

5 Modified Conceptual and Concrete Architecture

We discovered that the major subsystems interact with each other via Protocol. Hence, Protocol is a façade between the major subsystems, allowing greater independence for every subsystem.

DataNode has an important component, which is BlockManagement. BlockManagement is the component that communicate with NameNode for it's rack-awareness ability (used to determine the next location of the block replication), and send frequent updates (of heartbeats and blockreports) to NameNode.

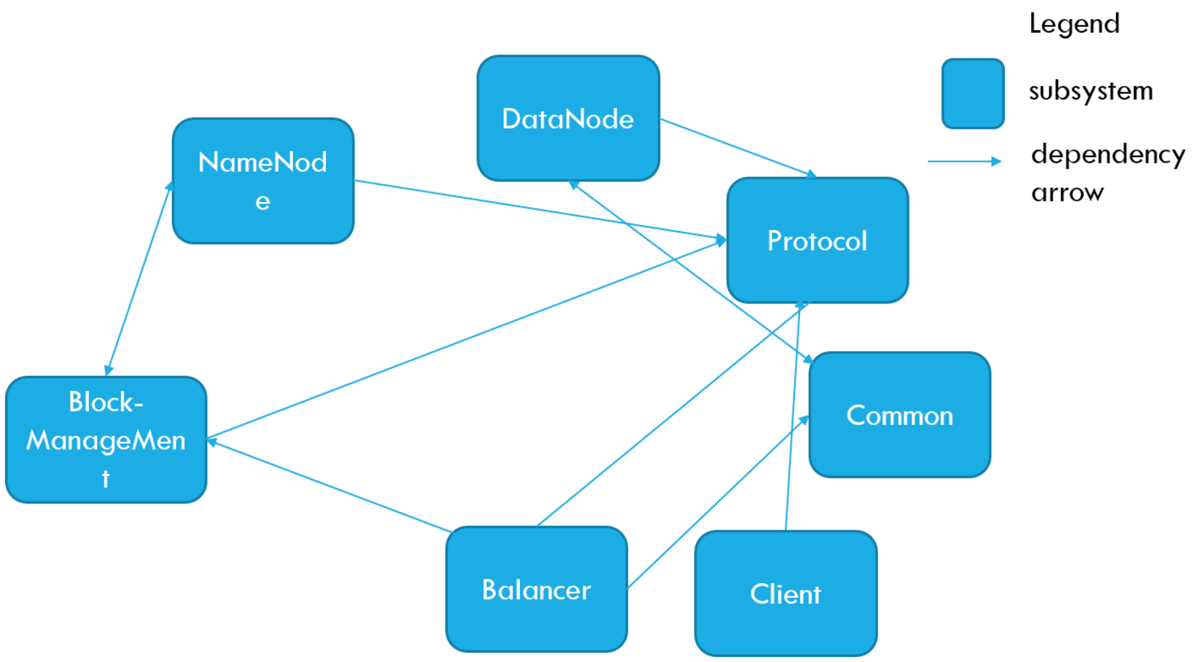

Figure 5: Modified and final concrete architecture of HDFS

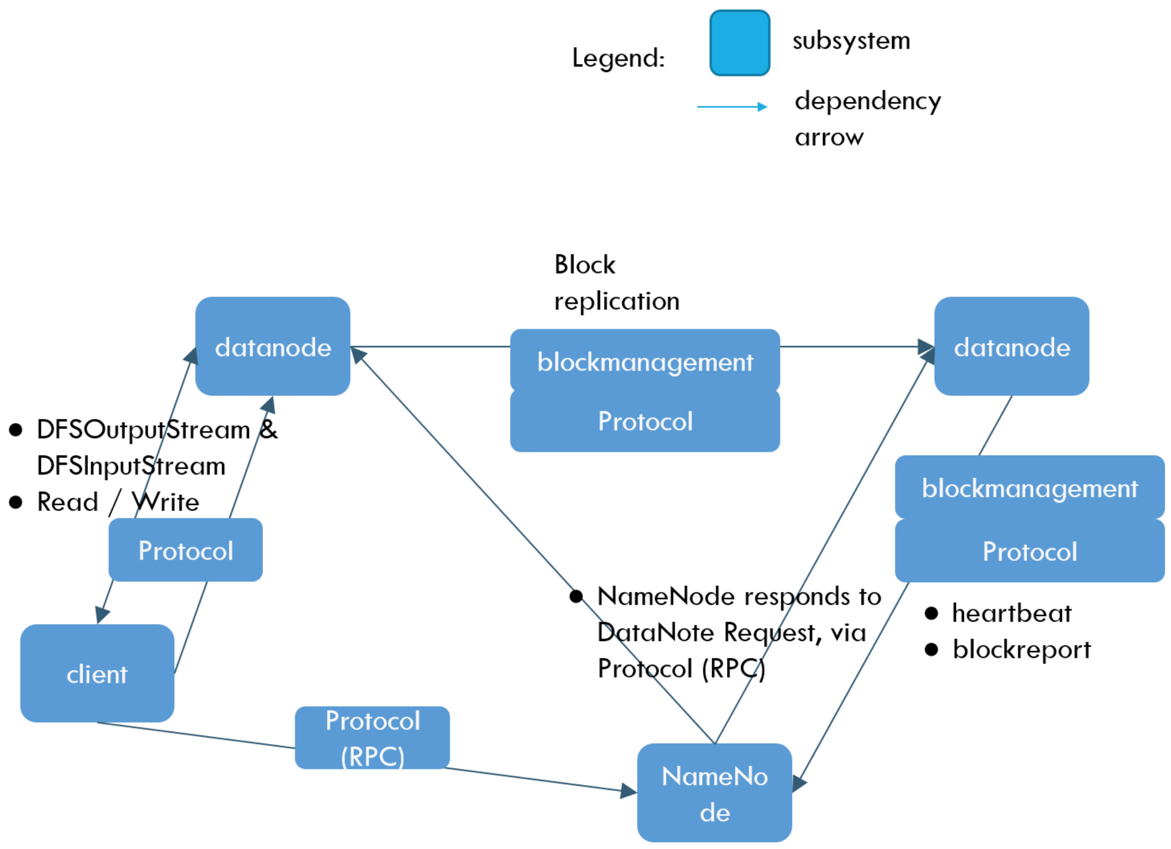

Figure 6: Modified Conceptual Architecture of HDFS after accounting for the extracted concrete architecture

6 Use Case

Figure (7) diagram describes the control flow of a client writing a file to HDFS. The client issues a write command, and HDFS successfully writes the file to its DataNodes. This Use Case shows how the subsystems DataNode and NameNode interact with each other, with the DFSOutputStream being a component within the Client subsystem. Note that the figure intentionally withholds detail because all of the implementation to write to a system would not fit in the diagram. Here are the steps:

1. The DFSOutputStream caches the file data into a temporary local file.
2. DFSOutputStream contacts the NameNode using the Protocol method addBlock to add a block to the file and return a pipeline of DataNodes for the client.
3. The Client writes the each DataNode in the pipeline, beginning with the first DataNode
4. On completion, the DataNode passes the remaining file to the next DataNode
5. Each DataNode sends a blockRecieved message to the NameNode and the DSFOutputStream

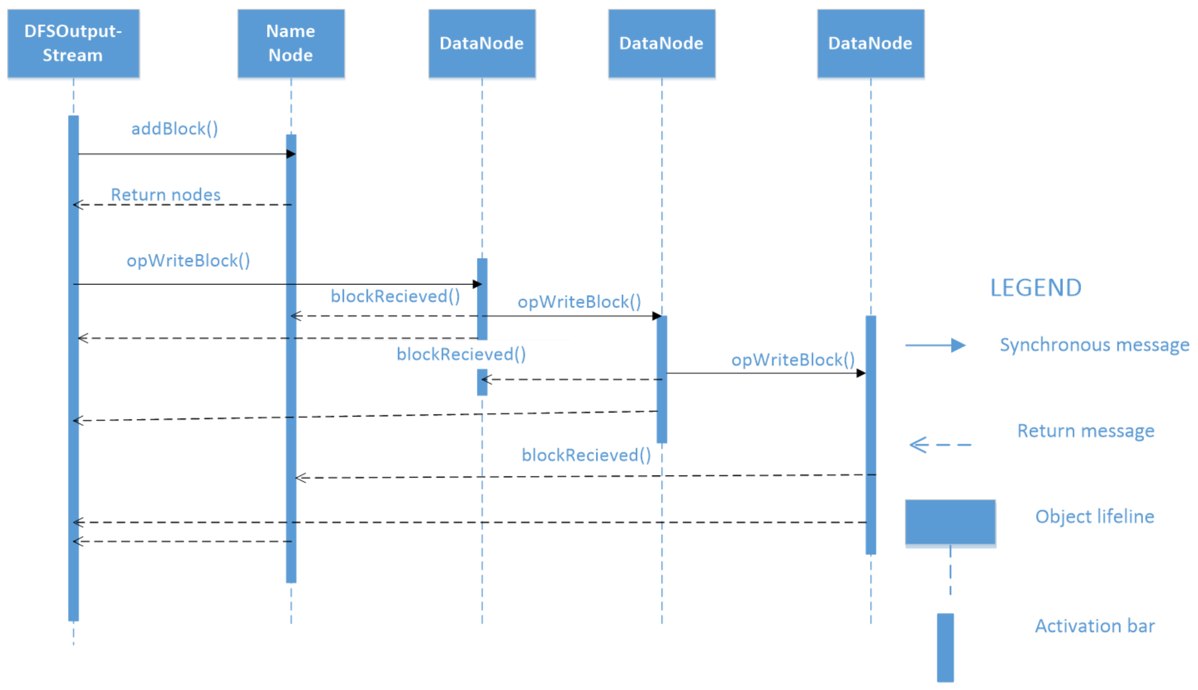

Figure 7: Sequence diagram describing the Use Case of writing to HDFS

7 Conclusions

Hadoop Distributed File System provides powerful and reliable storage system for Hadoop applications. The subsystems within HDFS have important dependencies with each other, and all these subsystems working together form the base of HDFS. Using LSEdit, Notepad++, and MicroExcel, we could extract the complete concrete architecture of HDFS, carefully analyze each dependency, then after repairing the architecture by either adding or removing dependencies, we finally produced the concrete architecture. Each subsystem needed to be analyzed to determine if one subsystem depends on another, as such each subsystem was described in detail about its purpose and how it relates to the conceptual architecture.

8 Lessons Learned

There are many lessons learned while extracting the concrete architecture of Hadoop HDFS. We realized that to retrieve the concrete architecture is a process that takes lots of effort and time using search and investigate technique. Using the dependency files and visualizations tools is time consuming but if used properly it gets the concrete architecture accurately. We also found that the concrete architecture varies a lot from the conceptual architecture as the concrete architecture contains a lot more dependencies than the conceptual one and this is expected as a consequence of implementation to get the expected functionality.

9 Data Dictionary

FSImage - a file that represents a point-in-time snapshot of the filesystem's metadata
LSEdit - a graph visualization tool that is particularly suited to exploration and editing of software
Heartbeat - receipt of heartbeat from a DataNode implies the DataNode is functioning properly.
Blockreport - a block report contains a list of all blocks replicated on a DataNode
MicroExcel - spreadsheet developed by Microsoft for Windows, macOS, Android and iOS
Notepad++ - a text editor and source code editor for use with Microsoft Windows
TCP
Understand - an analysis tool used to understand source code

10 References

[1] "Apache Hadoop 2.7.3," *Apache Hadoop*, 18-Aug-2016. [Online]. Available: https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/hdfsdesign.html. [Accessed: 01-Nov-2016].
[2]*Developer.com*, 2016. [Online]. Available: http://www.developer.com/imagesvr_ce/478/HDFS.jpg. [Accessed: 02- Nov- 2016].
[3]K. Shvachko, H. Kuang, S. Radia, R. Chasler, "The Hadoop Distributed File System", *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. *1-10*. May 2016.
[[4] Hadoop Distributed File System (HDFS) Architectural Documentation – Assumptions and Goals, unknown. [Online]. Available: http://itm-vm.shidler.hawaii.edu/HDFS/ArchDocAssumptions+Goals.html [Accessed: 01-Nov-2016]