2011

# HadoopT - breaking the scalability limits of Hadoop

Anup Talwalkar

# HadoopT - Breaking the Scalability Limits of Hadoop

by

**Anup Suresh Talwalkar**

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Science

Supervised by

Associate Professor Dr. Minseok Kwon
Department of Computer Science
Thomas Golisano College of Computing
and Information Sciences
Rochester Institute of Technology
Rochester, New York
January 2011

Approved by:

---

Dr. Minseok Kwon, Associate Professor
*Thesis Advisor, Department of Computer Science*

---

Dr. Rajendra K. Raj, Professor
*Committee Member, Department of Computer Science*

---

Dr. Stanislaw P. Radziszowski, Professor
*Committee Member, Department of Computer Science*

# Thesis Release Permission Form

Rochester Institute of Technology
Thomas Golisano College of Computing
and Information Sciences

Title:

HadoopT - Breaking the Scalability Limits of Hadoop

I, Anup Suresh Talwalkar, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

---

Anup Suresh Talwalkar

---

Date

# Dedication

*To my parents. . .*

# Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Professor Minseok Kwon for his continuous support of my study, for his patience, motivation, enthusiasm. His guidance helped me in all the time of research and writing this thesis. I am grateful to my committee members Professor Rajendra Raj and Professor Stanislaw Radziszowski for supporting me throughout my Masters and providing me encouragement, sound advice and good company.

I would like to thank my fellow labmates and classmates for making my stay here in Rochester so enjoyable.

Finally I would like to thank my friend Nishant Vijayakumar, for lending me a helping hand when needed, and providing me with a sounding board when I needed to talk things out.

# Abstract

The increasing use of computing resources in our daily lives leads to data generation at an astonishing rate. The computing industry is being repeatedly questioned for its ability to accommodate the unpredictable growth rate of data. It has encouraged the development of cluster based storage systems. Hadoop is a popular open source framework known for its massive cluster based storage. Hadoop is widely used in the computer industry because of its scalability, reliability and low cost of implementation.

The data storage of the Hadoop cluster is managed by a user level distributed file system. To provide a scalable storage on the cluster, the file system metadata is decoupled and is managed by a centralized namespace server known as NameNode. Compute Nodes are primarily responsible for the data storage and processing.

In this work, we analyze the limitations of Hadoop such as single point of access of the file system and fault tolerance of the cluster. The entire namespace of the Hadoop cluster is stored on a single centralized server which restricts the growth and data storage capacity. The efficiency and scalability of the cluster depends heavily on the performance of the single NameNode.

Based on thorough investigation of Hadoop limitations, this thesis proposes a new architecture based on distributed metadata storage. The solution involves three layered architecture of Hadoop, first two layers for the metadata storage and a third layer storing actual data. The solution allows the Hadoop cluster to scale up further with the use of multiple NameNodes. The evaluation demonstrates effectiveness of the design by comparing its performance with the default Hadoop implementation.

# Contents

# List of Figures

# Chapter 1

# Introduction

A key challenge faced by enterprises today involves efficiently storing and processing large amounts of data. Consider web search engines such as Baidu [19] which currently handles about 3 petabytes of data per week. Yahoo! uses its large datasets to support research for advertisement systems. Ebay, an e-commerce organization, has heavy storage usage of about 6 petabytes [18]. The practice of analyzing huge amounts of data for better understanding of consumer needs have motivated the development of data intensive applications and storage clusters. The primary requirement of these applications is a highly scalable, highly available and reliable cluster based storage system.

The MapReduce programming model [3] was developed by Google to meet the rapidly growing demands of their web search indexing process. These MapReduce computations are performed with the support of their cluster based data storage system known as the Google File System (GFS) [2]. The success of the Google File System and MapReduce inspired the development of Hadoop [8], an open source framework for building large clusters.

Typically, Hadoop is used as representative of the large scale storage system and the MapReduce programming paradigm because the potentially high performance implementation of Google is not publicly available. It has gained immense popularity because of its efficiency, scalability, cost effectiveness and publicly available distribution.

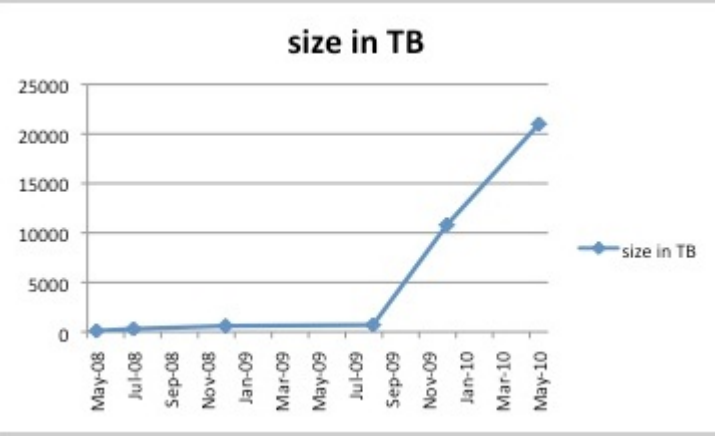


Figure 1.1: Data growth at Facebook [21]

Facebook [20], a popular social networking website with a user base of more than 500 million has shown remarkable data accumulation over the past few years. It is a strong supporter of Hadoop and has extensive use for its storage needs. Figure 1.1 shows the unprecedented growth rate of the cluster usage at Facebook.

Similar to the Google File System, the HDFS implementation follows the master/slave architecture [7]. The master node known as NameNode is the repository of the file system namespace and contributes in the primary metadata operations. Slave nodes known as DataNodes are responsible for the actual data storage. By separating the metadata, workload of the file system is dispersed among the NameNode and compute nodes improving the response time of the HDFS.

Despite the popularity of Hadoop, the effectiveness of the HDFS has been questioned in recent research considering its single NameNode architecture [5]. The architecture runs the potential risk of a single point of failure, as the failure of the NameNode causes the cluster to go offline. Also, the storage capacity of the Hadoop cluster is limited to the memory provided on the NameNode server. Thus, the Hadoop architecture imposes a substantial scalability penalty and fails to accommodate a significant data growth rate beyond its memory limitations.

We propose a solution involving multiple NameNodes for Hadoop with distributed metadata storage. We hypothesize that by distributing the metadata storage, the Hadoop file system will gain immense storage capacity by eliminating the bottleneck of the single NameNode. The architecture will provide the ability to dynamically add NameNodes as required, thereby increasing the underlying performance and scalability. The multiple NameNode architecture will also contribute to distributing the workload of the Hadoop file system and eliminate the single point of failure.

## 1.1 Our Contribution

This thesis contributes to the field of distributed storage systems in several ways. First, it explores the Hadoop architecture and focuses on identifying the limitations of its master/slave design. Second, it explores the possibilities of improving the scalability of the Hadoop storage system by eliminating the bottleneck at the NameNode. It introduces a new architecture for HDFS storage using a distributed namespace. Finally, it performs evaluation of the new architecture and discusses how removal of the bottleneck enhances the storage capacity and improves the fault tolerance of the cluster. The specific contributions are as follows:

#### Exploring the Hadoop Architecture and its Drawbacks

This thesis first analyzes and explains several drawbacks of the Hadoop architecture. It briefly discuss several issues involving the single point of namespace storage and how the Hadoop file system, instead of providing flexible massive storage capacity actually fails in scaling the storage with the upper bound on its metadata storage.

**Improving the Scalability with the Distributed Namespace Architecture**

This thesis is the first to analyze different possibilities of introducing multiple NameNodes in the Hadoop file system. HadoopT, a tree structured metadata storage is proposed in this work which allows extended storage capacity to the Hadoop file system. The details of design and implementation of HadoopT are briefly discussed in this work.

**Evaluating the New Architecture in Comparison with the Default Hadoop Configuration**

We will evaluate the new file system architecture by comparing it with the single NameNode architecture of Hadoop file system. The evaluation of the Hadoop file system is performed based on storage capacity, metadata storage capacity, MapReduce job execution and response to machine failure.

## 1.2 Organization

This thesis is organized as follows. Chapter 2 provides background into the Hadoop framework, HDFS architecture, and explains its drawbacks in relation with the single NameNode. Chapter 3 It discusses recent work that has been done for the improvement of scalability and fault tolerance of the cluster. Chapter 4 proposes a design for a new architecture, HadoopT, as distributed metadata storage solution for the Hadoop file system. Chapter 5 dives into the effectiveness of the architecture with its implementation details. Chapter 6 analyzes the performance of the new architecture in comparison with the singleton NameNode architecture of Hadoop and performs evaluation in terms of scalability, availability and performance. Finally, Chapter 7 concludes the thesis.

# Chapter 2

# Overview of the Hadoop Architecture

For the computer industry, effective and prompt analysis of gathered data means better understanding of consumer needs and more business. As the requirement of high storage capacity and data intensive computing grows, the scale of storage clusters increases. The key aspect of making such storage clusters cost effective and efficient is utilizing an appropriate software framework and platform for large scale computing.

Hadoop was chosen for this thesis for several reasons. First, it is popular and widely used by a number of leading organizations including, Amazon, Facebook, Google, Yahoo! and many others. Second, it is designed for commodity hardware, significantly lowering the cost of building the cluster. Third, Hadoop is an open source technology developed in Java, making it easier to obtain, distribute and modify whenever necessary. Its effective use, lower cost and easy access makes it a potentially stable base as a large scale storage system for future technologies. Thus, research into the architecture of Hadoop file system should help data intensive applications with their rapidly growing storage needs.

The Hadoop framework is written in Java which allows its portable installation across many available operating systems like Linux, freeBSD, Windows, Solaris and Mac OS/X. HDFS works as a data storage platform for many other client applications including Mahout - for machine learning algorithms, Hive - as a data warehouse engine and Zookeeper - a high performance coordination service [8]. Here, the architecture of Hadoop file system along with its metadata storage is described. Various drawbacks of the Hadoop architecture and

their effects on the data storage system such as scalability and availability limitations are discussed in this section.

## 2.1 Hadoop Cluster Architecture

Implementation of Hadoop is carried out in two main service components of the master/slave architecture. The file system metadata is decoupled from its actual data located on an individual NameNode machine. Decoupling provides flexibility to the architecture to accommodate more DataNodes in the cluster.

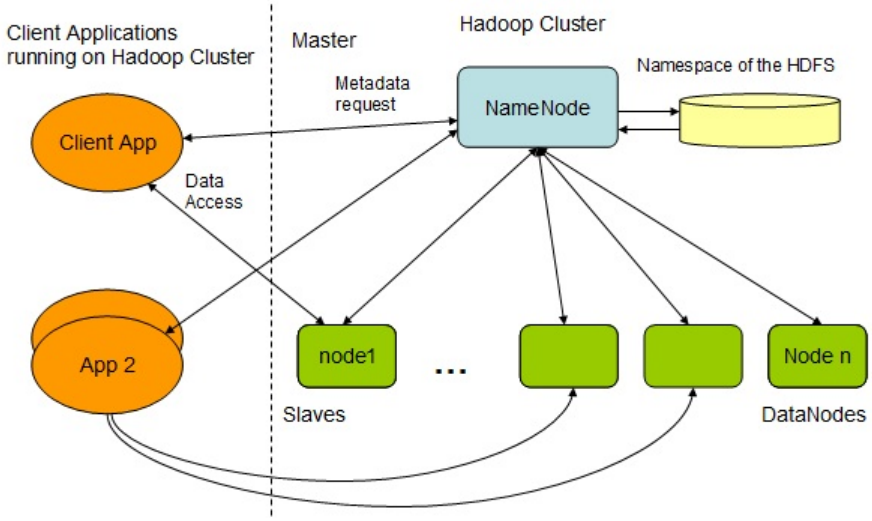


Figure 2.1: Hadoop Distributed File System (HDFS) Architecture

Figure 2.1 shows the master/slave architecture of Hadoop with client applications accessing the cluster. For data access, a client application communicates with the NameNode server to obtain the file system metadata. The data transfer then takes place between the client application and DataNodes. The decoupled architecture thus establishes minimum involvement of the single NameNode to reduce the workload on the cluster. For executing MapReduce jobs on the cluster, the jobs are submitted from the NameNode which are further distributed among the necessary DataNodes. The data is processed on the DataNodes as Map and Reduce tasks and the output is written back to the file system for client access.

## 2.2 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) [4] serves as the large scale data storage system. Similar to other common file systems, the HDFS supports hierarchical file organization. The NameNode splits large files into fixed sized data blocks which are scattered across the cluster. Typically the data block size for the HDFS is configured as 128MB, but it can be configured by file system clients as per usage requirements. The data storage is of type write once/read many (WORM) and once written, the files can only be appended and cannot be modified to maintain data coherency.

Since HDFS is built on commodity hardware, the machine failure rate is high. In order to make the system fault tolerant, data blocks are replicated across multiple DataNodes [7]. HDFS provides replication, fault detection and automatic data block recovery to maintain seamless storage access. By default replication takes place on three nodes across the cluster [8]. When a client tries to access the failed DataNode, the NameNode maps the block replica and returns it to the client. For achieving the high throughput, the file system nodes are connected by high bandwidth network.

### 2.2.1 NameNode

The NameNode maintains the file system metadata as the HDFS directory tree and operates as a centralized service in the cluster. It controls the mapping between file name, data block locations and the DataNodes on which data blocks are stored. It also writes the transaction logs to record modifications in the file system. Clients communicate with the NameNode for common file system operations such as open, close, rename and delete.

### 2.2.2 Namespace

The namespace is a live record of the HDFS located on the centralized NameNode server. It is a directory tree structure of the file system which documents various aspects of the

HDFS such as block locations, replication factor, load balancing, client access rights and file information. The namespace serves as a mapping for data location and helps HDFS clients to perform file system operations.

The metadata is stored as a file system image (fsimage) [7] file which is a persistent checkpoint of the file system. The edit log records the write operations submitted by the file system clients. When the edit log size exceeds a predefined threshold, the NameNode moves the transactions into live memory and apply each operation to the fsimage.

A backup of the namespace is periodically stored on the local disk of the NameNode and synchronized with a secondary master node as a provision against NameNode failure. When the NameNode reboots, it collects the file system namespace from the local copy.

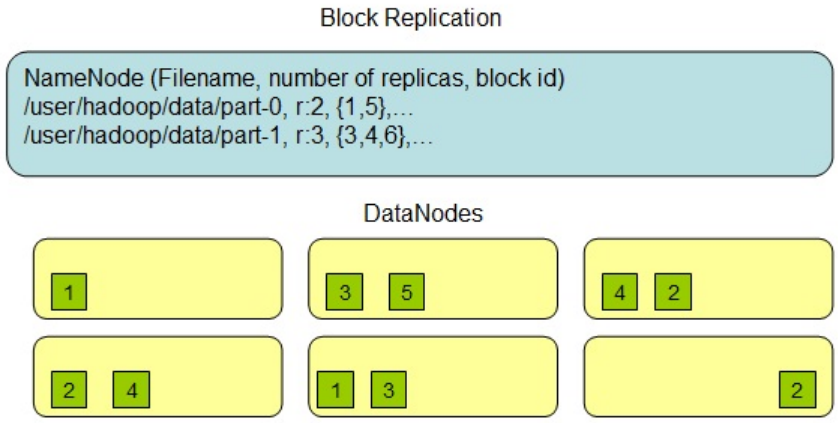


Figure 2.2: HDFS DataNodes

### 2.2.3 DataNode

A DataNode is a storage server that accepts read/write requests from the NameNode. DataNodes store data blocks for local or remote clients of HDFS. Each data block is saved as a separate file in the local file system of the DataNode. The DataNode also performs block creation, deletion and replication as a part of file system operations. For keeping the records up-to-date, the DataNode periodically reports all of its data block information to the NameNode.

DataNode instances can talk to each other for data replication. To maintain its live status in the cluster, it periodically sends heartbeat signals to the NameNode. When the NameNode fails to receive heartbeat signals from the DataNode, it is marked as a dead node in the cluster.

### 2.2.4 MapReduce

When MapReduce jobs are submitted to the cluster, the NameNode forwards them to appropriate DataNodes where the data resides. Upon receiving the jobs, the DataNode executes the job on its local system and returns the result. The task execution is handled by the MapReduce framework.

In the MapReduce programming model, the computations are divided into map and reduce tasks. The tasks are simultaneously performed on the DataNodes. In the mapping task, the data is processed into <key, value> pairs with a minimal coordination of DataNodes. In the reducing task, each output from DataNodes is combined to produce single output for the application.

## 2.3 Architectural Drawbacks

Regardless of the high storage capacity of the Hadoop file system, it has several limitations. The NameNode server of the Hadoop cluster is in possession of the entire storage metadata and performs all the important file system operations.

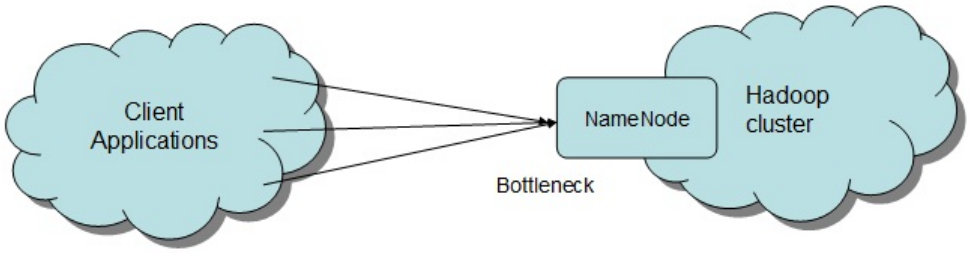


Figure 2.3: Single point of connection

For a distributed file system like HDFS, a centralized server maintaining the key information is a bottleneck in the architecture. This bottleneck establishes practical limits of growth on scalability, availability and performance of the system. We describe each of these limitations as follows:

### 2.3.1 Scalability Limitations

The centralized NameNode server stores the entire Hadoop file system namespace in live memory for faster access. As a result, the storage capacity of the cluster cannot grow beyond the available free memory space on the NameNode. Shvachko [5] estimates that the ratio of memory usage to the data storage is approximately 1GB memory per petabyte of data in the cluster or approximately 1,000,000 data blocks in the HDFS [5, 10]. In order to accommodate data referenced by a 100 million files, the HDFS cluster needs 10,000 nodes with eight 1TB hard drives. The total storage capacity of such a cluster is 60 petabytes. For this kind of scale of data, the NameNode needs to be installed on a much efficient server that can support 60GB of memory. With these estimations, HDFS cluster with a NameNode of 60GB memory space can not store data more than 60 petabytes.

With the current architecture, the entire Hadoop cluster relies on the performance of the single NameNode and its capacity to handle the namespace. For a growing cluster, increasing number of DataNodes means increasing workload on the NameNode. Thus, the increasing data load with memory limitations on the NameNode imposes scalability limits on the architecture.

### 2.3.2 Performance Limitations

Some of the vital tasks that the NameNode performs include receiving heartbeats and data block updates from DataNodes. The NameNode is responsible for providing all the necessary metadata information of data blocks stored in the cluster to its clients. For a rapidly growing cluster with increasing number of data blocks, its performance may reduce after

reaching the threshold of handling the file system metadata. During the time of peak usage of the cluster, the single NameNode may appear as a bottleneck in the cluster.

### 2.3.3 Availability Limitations

Use of the centralized NameNode server for serving the HDFS clients makes the system vulnerable to failure. In an event of NameNode failure, the cluster remains offline until it is fully recovered from the crash. When the NameNode recovers from the failure, it is restored to the state prior to the crash with its backup namespace image on the local storage. If the NameNode machine fails, a new NameNode can boot up with the help of the secondary NameNode. The recovery time is dependent on the cluster size and metadata size. Recovery may take longer time for large clusters due to large size of the backup namespace.

### 2.3.4 Isolation

Hadoop does not provide process or data isolation to its clients. A client application cannot control the data storage or set the number of nodes to perform computations. As a result, a client application has access to the entire cluster. This is not suitable for organizations which accommodate diverse applications to perform job executions on their storage clusters.

# Chapter 3

# Related Work

Since the weakness of the centralized namespace storage of Hadoop has surfaced, there have been a few attempts and papers published providing different strategies for eliminating the single point of failure and increasing the storage capacity of the architecture. The development of project entitled, "AvatarNode" [15] provides a solution on the issue of the single point of failure of Hadoop. The paper was published by Feng Wang, *et al.*[16] from IBM China Research Laboratory; entitled "Hadoop High Availability through Metadata Replication" discusses improved failover of the NameNode. The last two sections discuss the work which has been done to improve the storage capacity of the cluster. The paper by George Porter [11], "Decoupling Storage and Computation in Hadoop with SuperDataNodes" provides a solution to increase the storage capacity of the cluster.

## 3.1 AvatarNode

The AvatarNode [15] was developed as a faster failover mechanism to deal with the single point failure of the NameNode. The primary AvatarNode runs exactly same as the NameNode and writes its transaction logs into the shared NFS filer. If another instance of AvatarNode is instantiated, it runs in standby mode. The standby AvatarNode encapsulates an instance of the NameNode and secondary NameNode. It keeps reading the transaction

logs from the same shared NFS filer and keeps feeding the transaction logs to the encapsulated NameNode instance. The NameNode within standby AvatarNode is kept in safe mode to prevent it from participating in the cluster activities.

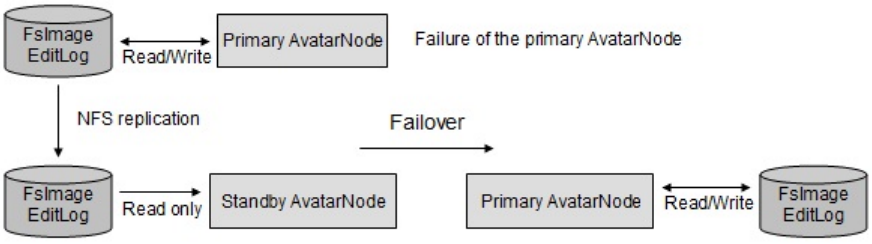


Figure 3.1: Standby AvatarNode taking place of pimary AvatarNode during the failover process

HDFS clients are configured to access the AvatarNode via Virtual IPAddress (VIP). If the primary AvatarNode fails, the failover is performed by switching the VIP to the standby AvatarNode. As the clients receive the entire data block list and replica locations at the time of file open operation, file read operations are not affected by the failover period. However, if the file is being written during the time of failover, client receives an I/O exception after the failover event. The failover does not affect MapReduce task execution as the framework is designed to retry any failed tasks.

**Discussion**

The AvatarNode is effective against NameNode failures and keeps the namespace data protected, thereby actively participating in the cluster usage. However, the AvatarNode does not improve scalability of the architecture and still has the single point of access to the cluster.

## 3.2 High Availability through Metadata Replication

There are several ways to deal with single point of failure in Hadoop. Feng Wang′s paper [16] discusses the metadata replication based solution. The solution consists of three major

phases: the first is the initialization phase which initializes the execution environment of high availability; the second is the replication phase which replicates metadata from critical node to corresponding backup node at runtime; and the third is the failover phase which resumes the running of Hadoop despite the critical node being out of work. As the file system information and EditLog transactions are stored as a backup copy on the NameNode, the solution provided in this paper only concentrates on the replication of critical metadata such as leases states of file writes.

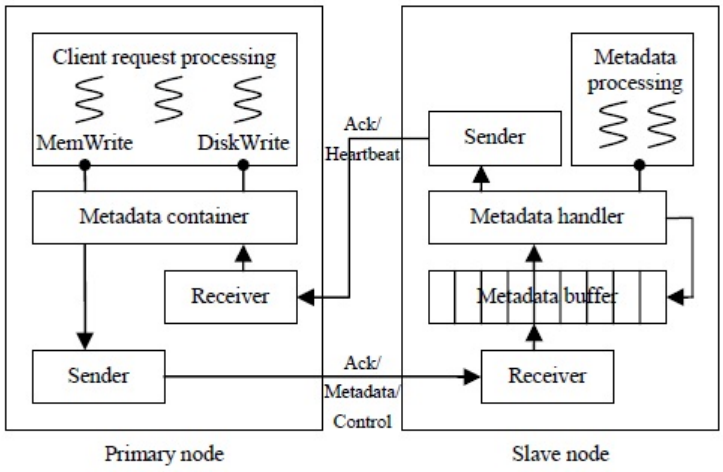


Figure 3.2: Architecture for replication [16]

In the initialization process, multiple slave nodes register with the primary node for the up-to-date metadata information. The second stage of replication resumes after initialization of the slave nodes. Figure 3.2 explains the architecture designed for critical metadata replication. The primary node collects metadata from client request processing threads and sends it to the slave node. The slave node handles the processing of the received the metadata which includes in memory and in disk processing of clients requests. The slave node sends a heartbeat signal to the primary node to keep track of its live status.

In the failover state, when slave nodes fails to receive the acknowledgement of its heartbeat message, leader election algorithm is used to decide which slave node will take place for the primary node. Upon selection of a slave node as the primary node, it changes its IP address to the IP address of the retired primary node, thus finishing the failover process.

#### Discussion

This paper presents an adaptive method for failure recovery of the NameNode by metadata replication with lowered failover duration, but it does not solve the issue of single point of failure with Hadoop. The solution is suitable for medium amounts of files but not for higher amounts of I/O requests. Also, the metadata replication does not improve the scalability of the architecture.

## 3.3 Improving NameNodes Scalability

The Hadoop RPC Server implementation [13] has a single listener thread that reads data from the socket and puts them into a call queue for the NameNode threads. NameNode gets to process the RPC requests only after all the incoming parameters copied and deserialized by the listner thread. For a heavy load cluster, a single thread operating the RPC listener tasks is a bottleneck causing the clients receiving RPC socket errors. Due to this bottleneck, clients are unable to utilize power of the NameNode.

It is observed that most workload of the NameNode appears from read/write operations of the data. The NameNode metadata management was enhanced by creating a pool of RPC reader threads which works to deserialize the RPC requests from the clients. Also, most of the file system operations are read only and do not trigger any synchronous transactions. By changing the current FSnamesystem lock to readers-writer lock, the performance of the NameNode improved significantly.

**Discussion**

The solution improves the performance of a heavy usage Hadoop cluster by improving the bottleneck of RPC listener thread. The solution was effective in improving the performance of the NameNode to handle heavy workload, but it fails to provide a solution to the linear scalability and single point of failure of the NameNode. In the long run, the cluster will not be able to accommodate the data growth.

## 3.4 Decoupling Storage and Computation with SuperDataNode

Porter [11] discusses the use of a decoupled DataNode architecture to provide increased data storage and computation in Hadoop. The paper introduces SuperDataNodes which are servers containing more disks than the regular nodes in Hadoop. It can host amounts of data equivalent to the storage capacity of many DataNodes. The design is a storage-rich architecture of Hadoop.

Figure 3.3 shows a SuperDataNode with a storage pool made up of several disks. From this pool, several file systems are built with the help of virtual machines with its own network interface assigned. The SuperDataNode is much richer than average storage layer with large magnitude of disks and network bandwidth. Each virtual machine forms a separate DataNode in the network where the jobs are executed individually by separate task tracker servers.

For storage of N bytes in the cluster with c being an average storage capacity of a single DataNode, the total number of DataNodes required to form the Hadoop cluster is N/c. The use of SuperDataNode decouples the amount of storage in HDFS from the number of DataNodes providing increased storage capacity with less number of DataNodes. Also, by separating the storage from task trackers, the task tracker servers can be turned off when not in use in order to lower the power consumption of the cluster.

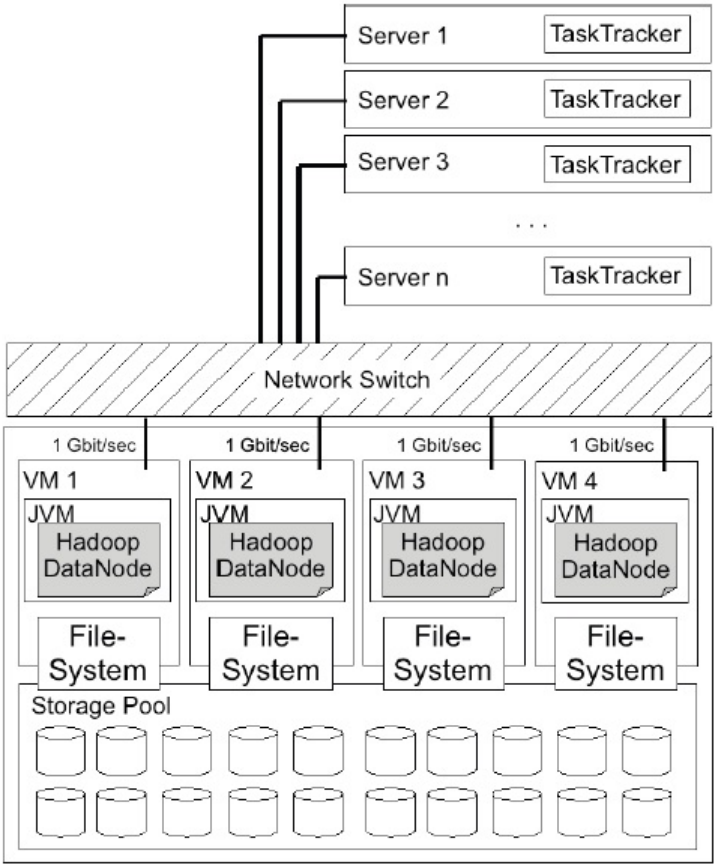


Figure 3.3: Architecture of a SuperDataNode [11]

**Discussion**

Since a single SuperDataNode accommodates data worth many DataNodes, its failure has significant impact on the storage. The use of SuperDataNode has no impact on the meta-data storage. As a result, it does not improve scalability of the architecture. The use of SuperDataNodes is not a cost effective solution to improve the storage capacity. Also, the network bandwidth of the architecture is affected due to single point of access to a large amount of data.

# Chapter 4

# HadoopT

In the previous chapters, we described drawbacks of the Hadoop architecture and the limitations it imposed on the growth of the cluster. Keeping in mind the prior work which has been done to overcome the limitations, it has led us to investigate into the possibility of a multiple master architecture for Hadoop. The new revised architecture of Hadoop first should scale up without the underlying limitation on metadata storage. Second, it should eliminate single point of failure by distributing the workload among multiple NameNodes. Finally, the architecture should build up a highly available system with performance equivalent to the default Hadoop architecture.

We propose a novel architecture for Hadoop file system; we call it HadoopT. The goal and focus of this work is to introduce multiple NameNodes in the architecture of Hadoop. The design of HadoopT must provide not only the distribution to eliminate the bottleneck of the single NameNode server, but should also take into account the core functionalities supported in Hadoop. Here, we describe the design details of HadoopT and present the development that is carried out to incorporate Hadoop file system operations. We will also discuss how the system should respond to the scalability, fault tolerance and availability issues.

## 4.1 HadoopT

The work presented in this section covers the core design of HadoopT. In the design of HadoopT, the cluster establishes a layered organization of metadata storage with three layers of nodes. Its metadata is divided across two layers of namespace servers to distribute the workload. Forming a tree structure with three layers, each layer participates in a certain type of metadata storage which can be mapped to the layer below. The clients are able to access the file system with the topmost layer of the architecture. The discussion section covers the systems performance with respect to primary Hadoop operations such as replication, fault tolerance, load balancing and MapReduce job execution.

### 4.1.1 Design Overview

Unlike the default Hadoop configuration where a single master node connects to multiple slaves, the design splits the centralized master component into two layers as SuperNode and NameNode with DataNode being the data storage layer. This additional decoupling of the master node gives the scope of using multiple NameNodes. Each layer in the design manages a specific type of metadata in the file system and holds a mapping with nodes in the layer below.

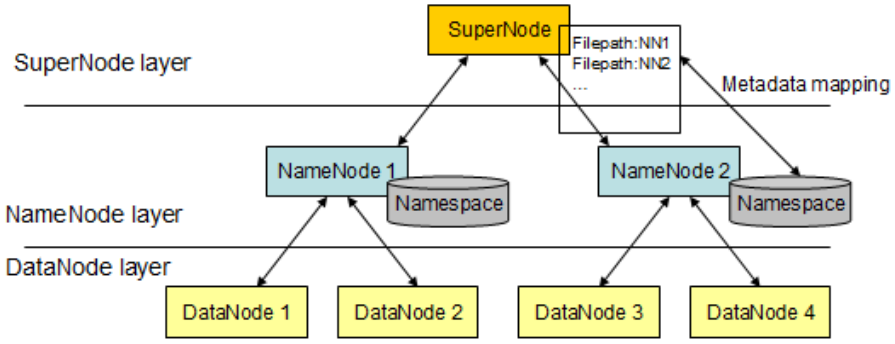


Figure 4.1: High level overview of the HadoopT architecture with 4 DataNodes, 2 NameNodes and 1 SuperNode

Figure 4.1 explains the cluster design of HadoopT with multiple NameNodes. The SuperNode is at the topmost layer of the architecture which controls all the NameNodes in the cluster. The second layer of the design is a pool of NameNodes which holds the file mapping from the SuperNode. Each NameNode in the NameNode layer manages a cluster of DataNodes assigned to it. Data transfer operations, Hadoop command executions and job submissions are performed through the SuperNode and forwarded to the respective NameNodes.

### 4.1.2 SuperNode

The SuperNode is a service in the cluster operating on a single node. Clients contact the SuperNode in order to perform file system operations such as open, close, read and delete. Besides regular file system operations, SuperNode provides a command line console to its clients to query the file system for data transfers and job executions. The SuperNode does not store any HDFS metadata in its local storage, but keeps all the metadata in live memory for faster access. It maintains a mapping between filenames and their location on the NameNode layer.

The SuperNode is the key component in the design of HadoopT cluster. It is a loosely connected entity in the cluster which manages high level metadata of the file system. The SuperNode communicates with the NameNode layer in the HadoopT cluster for controlling its file system operations. It periodically receives the active file record from the NameNodes to synchronize its metadata in the cluster.

On every boot up, the SuperNode acquires the necessary information from NameNodes configured in the cluster. The light weight memory storage makes the design suitable for metadata replication and effective against node failures. The primary operations of the SuperNode are to receive commands from the user, submit MapReduce jobs to the cluster and keep track of high level metadata of the distributed file system.

### 4.1.3 Set of NameNodes

NameNode is a service in the design of HadoopT which provides low level metadata storage of the cluster. Similar to the SuperNode, NameNodes store the metadata in live memory for faster access. The primary responsibility of NameNodes is to receive commands and MapReduce jobs from the SuperNode to execute on the set of assigned DataNodes.

The metadata of the NameNode includes the namespace directory tree, data block replication, block location and information of the DataNodes assigned to it. The NameNode also maintains the file system information, storage capacity of DataNodes and keeps track of all the live nodes in the cluster. Clients are able to access the file system and the HadoopT cluster exactly the same way as the default configuration of Hadoop. On the SuperNode service initialization, the NameNode layer responds to the request of metadata synchronization by providing information of all the files in the cluster.

### 4.1.4 Mapping

The SuperNode is loosely connected with the NameNode service by means of file system mapping and connection information. The file system mapping is a data structure storing all the paths of files which are available on the cluster and their locations on the NameNode. The connection information is the information with which the SuperNode is able to communicate with NameNodes for performing file system operations.

The SuperNode periodically sends requests to all NameNodes to update the file information. It uses a polling mechanism to keep the file path and NameNode mapping information up-to-date. Polling also works as a heartbeat mechanism between NameNodes and the SuperNode. i.e., if a NameNode fails, failed heartbeat acknowledgements are reported to the SuperNode. The SuperNode then removes the failed NameNode from its mapping. When the failed NameNode recovers from the crash, its mapping is restored on the SuperNode.

### 4.1.5 Client Access

Client applications are able to access data on the HadoopT cluster using two network hops. The data I/O operations are carried out from the SuperNode.

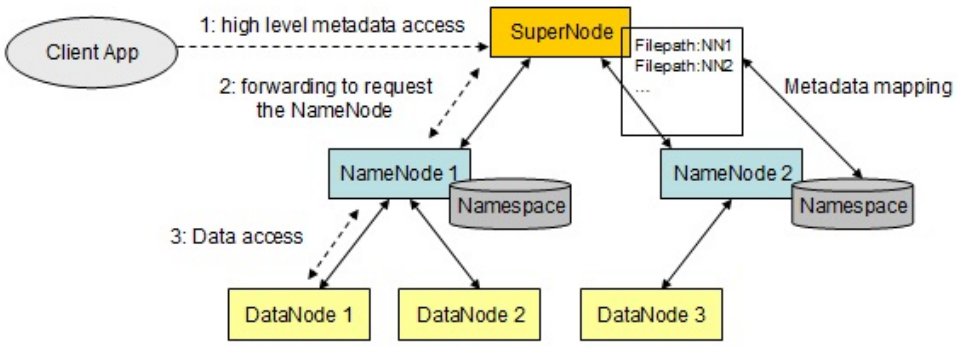


Figure 4.2: Client accessing files on the HadoopT cluster

First, client applications use the higher level namespace of the distributed file system to identify the NameNode that stores the low level metadata. Fileinfo mapping on the SuperNode forwards the access request to the appropriate NameNode. On the second hop, the NameNode retrieves data from DataNodes.

### 4.1.6 Data Distribution

The SuperNode finds an appropriate NameNode from its NameNode mapping when a user wants to access the distributed file system for read/write purpose. The I/O operations are then performed on the cluster by forwarding the request to the appropriate NameNode. Once the I/O request is executed, the SuperNode updates the file system metadata for successful modifications.

## 4.2 Discussion

### 4.2.1 Load Balancing and Data Locality

The SuperNode keeps track of the disk space utilization of each NameNode. When clients create folders on HadoopT, the SuperNode chooses an appropriate NameNode to balance the data storage. The folders cannot be duplicated across NameNodes in order to preserve the unique file path in the tree structure. Files stored in a single folder are maintained on the same NameNode to preserve data locality. Clients also have an authority on deciding the secondary layer NameNode for data storage.

### 4.2.2 Replication

As the SuperNode does not save the data structure mapping on its disk, its boot up requires initial communication with all the NameNodes. Replication can be achieved by invoking multiple instances of the SuperNode. Multiple SuperNodes synchronize with each other depending upon their polling response time with all the NameNodes.

For more than two NameNodes, clients are able to set the replication factor for the files. Files are mapped on multiple NameNodes according to the replication factor. Their mapping is recorded on the SuperNode as primary and secondary host NameNode. If a NameNode fails, the files can be accessed from their secondary replica.

The HadoopT file system storage is of type write once read many (WORM). This makes the metadata replication easier. The file system modifications are simultaneously performed on all the replicas.

### 4.2.3 Fault Tolerance

The HadoopT file system is fault tolerant. While the low level data storage is backed up by data block replication, NameNodes metadata is also replicated on a secondary NameNode for fault resistance. Failure of a NameNode in HadoopT signifies partial failure of the

cluster while rest of the cluster is still accessible to its users. Failure of the SuperNode can be nullified by executing multiple instances to replicate the high level metadata. Multiple instances of the SuperNode also overcome the issues of single point of failure.

### 4.2.4 Job Execution

When a client submits a job to the SuperNode, it gets mapped to the appropriate NameNode. The DataNodes managed by the NameNode executes the job as a MapReduce task. The output of the MapReduce task is stored as a file in the distributed file system. As a result, clients are able to access the output using the SuperNode like any other file in the cluster.

# Chapter 5

# HadoopT Architecture and Implementation

In the previous chapter we discussed our proposed design for metadata distribution, HadoopT. An important consideration that must be taken into account while implementing the HadoopT design is its integration with the underlying Hadoop framework. The HadoopT design discussed in the previous chapter provides basic structure needed for the actual development. The primary components of this implementation are SuperNode service, communication between the SuperNode and the NameNode layer and support to the file system operations. Here, we detail the development of the design and its integration with the Hadoop implementation.

The architecture is developed in Java 1.6 for portability across several platforms such as Linux, Windows and Mac OS/X. The Java implementation also provides its seamless integration with the current Hadoop implementation. For this system, SFTP protocol is used for the data transfer operations and the communication is handled using XML-RPC protocol.

## 5.1 Overview of the architecture

The HadoopT architecture is implemented in two main components, the SuperNode and HadoopT communication server. These components are built on the basis of XML-RPC communication protocol for message transfer and SFTP protocol for the data exchange. Besides these components, the key factors of this implementation are the data structures

which are built for mapping the file system metadata.

The HadoopT server is an XML-RPC server initialized on the NameNode layer for communication with the SuperNode. The HadoopT server receives request from the SuperNode as a RPC call and executes requests on the NameNode. The result of the execution is returned to the SuperNode as a return object for the RPC call.

The SuperNode is a service initialized on a separate server for communication with the NameNodes through HadoopT server. The SuperNode stores high level metadata of the file system as a fileinfo mapping and monitors a set of NameNodes configured in the cluster. The SuperNode interacts with all the NameNodes using XML-RPC communication channel by invoking the XML-RPC client objects initialized on the HadoopT server.

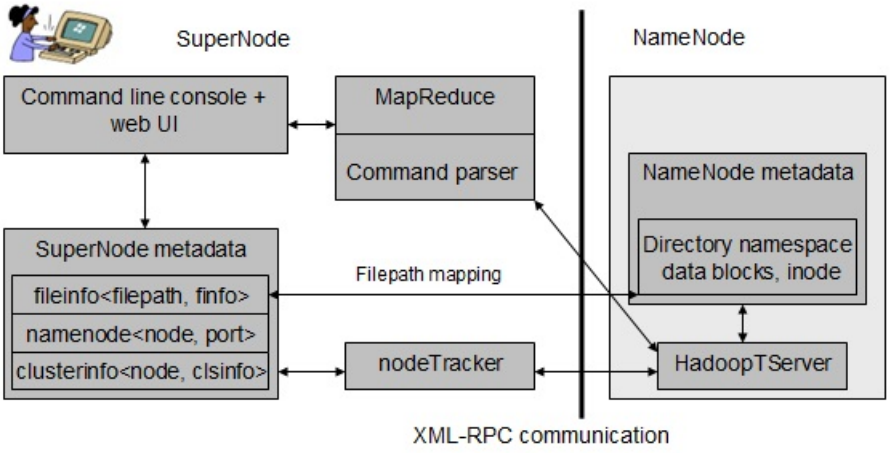


Figure 5.1: Implementation of the HadoopT architecture

Figure 5.1 describes the design of HadoopT implementation. The following sections discuss each of the components with their roles and interactions in the system.

### 5.1.1 System Interaction

The system is designed to distribute the workload and the namespace storage of the NameNode. The SuperNode can connect to one or more NameNodes by means of XML-RPC communication.

The XML-RPC client server model is implemented to overcome the fixed connection overhead on the system. Instead of keeping an open socket, the command line and data transactions are performed by opening the connection for the duration of the request execution. The SuperNode holds XML-RPC client objects for the XML-RPC server on NameNodes to invoke the necessary methods. When the SuperNode is not being used, the system stays in sleep mode allowing other clients to use the bandwidth with higher throughput. The key components of the system interaction are as follows:

**HadoopTServer**

It is an XML-RPC server component of the architecture. It starts the web and RPC server to establish the communication protocol between the NameNode and the SuperNode. The HadoopTServer also holds a local copy of the fileinfo data structure to keep in sync with the SuperNode.

**nodeTracker**

The nodeTracker is a separate thread that runs on the SuperNode. It keeps track of all the NameNodes in the cluster using the namenode data structure. The nodeTracker thread repeatedly invokes methods on the HadoopT server to keep the SuperNode metadata up-to-date.

**NameNode Objects**

NameNode objects are XML-RPC client objects for the HadoopT server. They implement wrapper methods for the RPC calls made by the SuperNode. With this, the SuperNode connects with the HadoopT server by means of supernodeProtocol class implemented on the HadoopT server. NameNode objects are invoked by the SuperNode to execute commands on the cluster.

### 5.1.2 SuperNode - HadoopT Communication

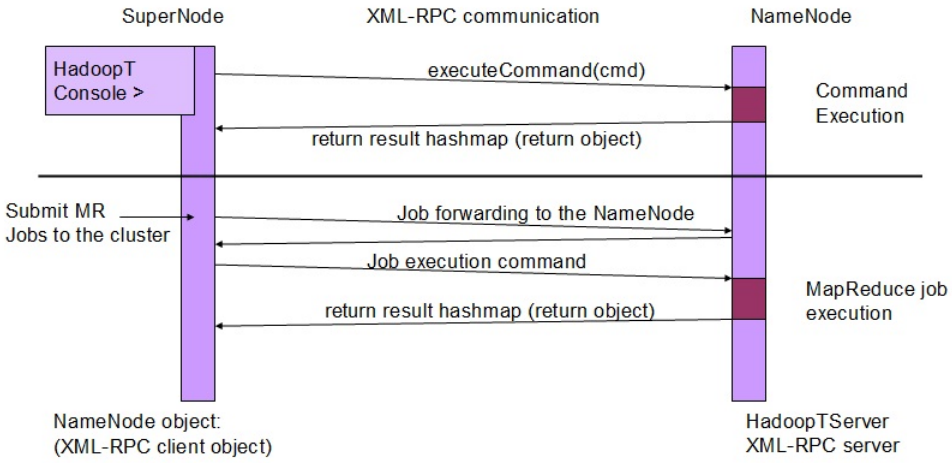


Figure 5.2: SuperNode-HadoopT communication

Figure 5.2 illustrates the system interaction by following the flow of data access and MapReduce job execution.

1. The client requests the SuperNode for the NameNode object to invoke for accessing the location of the required file. If no location is found, the SuperNode returns the request to the client.

2. The client can read files on the cluster by executing commands such as tail/cat on the SuperNode. The result of the execution on mapped NameNodes is returned back to the SuperNode console.

3. For data access, the SuperNode first finds the specific NameNode location of the file from its data structure mapping. It then executes the command for file transfer using simple Hadoop get/put arguments. The data is then exchanged between the SuperNode and the NameNode using SFTP protocol to complete the execution.

4. To execute MapReduce jobs on the cluster, the client pushes all the required class files to the cluster. The SuperNode finds and invokes fileinfo specific NameNode objects to forward MapReduce jobs to appropriate NameNodes.

5. The NameNode executes the MapReduce job using available DataNodes and the result is stored in the cluster as an output file in a folder specified by the client.

6. A separate thread called nodeTracker periodically checks newly created files and updates the SuperNode metadata accordingly. With the updated file system information the client can access the result of the MapReduce job execution.

The loosely connected system addresses the problem of network disruption and makes it fault tolerant. As a result, the architecture is flexible towards accommodating multiple instances of the SuperNode.

## 5.2 Metadata Management

### 5.2.1 Data Structures

The SuperNode holds three primary data structures: namenode, fileinfo and clusterinfo. These data structures are the building blocks of SuperNode metadata. The primary cluster information is stored on the namenode and clusterinfo tables and fileinfo handles the file mapping information in the HadoopT cluster.

```
/** File Information mapping of: file name - nodeURL*/
public static Hashtable<String, FileInfo>
fileInfo = new Hashtable<String, FileInfo>();
/** Cluster information mapping of: nodeURL and ClusterInfo object*/
public static Hashtable<String, ClusterInfo>
clusterinfo = new Hashtable<String, ClusterInfo>();
/** Data structure for storing all the NameNode objects*/
public Hashtable<String, Namenode>
namenode = new Hashtable<String, Namenode>();
```

Figure 5.3: Data Structures

**namenode**: The namenode data structure stores the mapping of the NameNode URL to the NameNode object. It is used to obtain the required NameNode object for the SuperNode. The SuperNode polls this data structure to track all the live NameNodes in the cluster by a heartbeat mechanism. The HadoopT cluster can be reconfigured by modifying this data structure.

**fileinfo**: The fileinfo data structure maintains the mapping of the file path to the fileinfo object. The mapping locates the NameNode on the second layer. The mapped fileinfo objects are used to access NameNodes on the cluster, where the files are actually stored.

**clusterinfo**: The clusterinfo data structure stores basic information about the cluster such as storage size, available space and number of DataNodes that a NameNode manages. The clusterinfo object is useful for clients to identify empty space on each NameNode and determine an appropriate NameNode for their data storage.

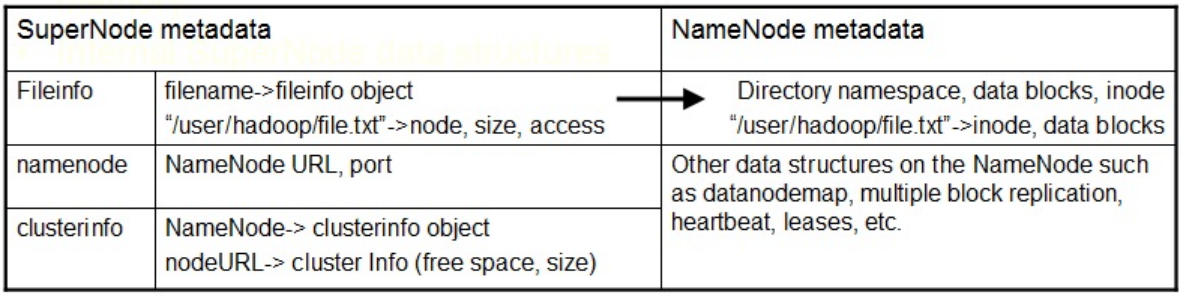


Figure 5.4: SuperNode-NameNode Metadata mapping

The SuperNode invokes the fileinfo and namenode data structures whenever a client needs to access files on the HadoopT cluster. Using fileinfo and namenode, the command submitted on the SuperNode console is parsed and mapped to the appropriate NameNode object. The object is then returned to the client to perform all the necessary file system operations. After the command execution, the result is returned to the SuperNode client.

The clusterinfo data structure is periodically updated by the nodeTracker to keep the SuperNode aware of available free space on the cluster. It holds the key information regarding free space available on each NameNode controlled by the SuperNode. The folder

creation and file storage operations depend on the free space available on the NameNode layer.

### 5.2.2 Replication

On a distributed system like HadoopT, there is always a possibility of machine failure. The failure can either be of a SuperNode, NameNode or DataNode machine. HadoopT uses a file replication method to avoid the file unavailability on the cluster. If replication is enabled by the SuperNode client, two different NameNodes are chosen by the SuperNode to store the file. The secondary NameNode information is kept in the fileinfo data structure of the SuperNode.

The nodeTracker thread keeps track of the low level file system metadata on the NameNode. While periodically checking the metadata, the file replica is treated as the original file and the SuperNode namespace is overwritten. If a NameNode becomes unavailable, its metadata information is removed from the SuperNode. The nodeTracker then replaces the file information by the metadata of its replica. For this transition period, the file may become unavailable on the SuperNode. The data replication method protects the HadoopT cluster from data unavailability.

## 5.3 SuperNode Operation

The SuperNode executes all the metadata operations. It also manages the high level file system and periodically syncs the second layer NameNode metadata with the data structure it holds. The file system clients are able to view and access the file system through the console provided by the SuperNode. The console is useful to execute the SuperNode commands and perform MapReduce jobs on the cluster. The SuperNode also makes the file placement decisions based on the amount of free space available on each NameNode.

### 5.3.1 SuperNode Startup

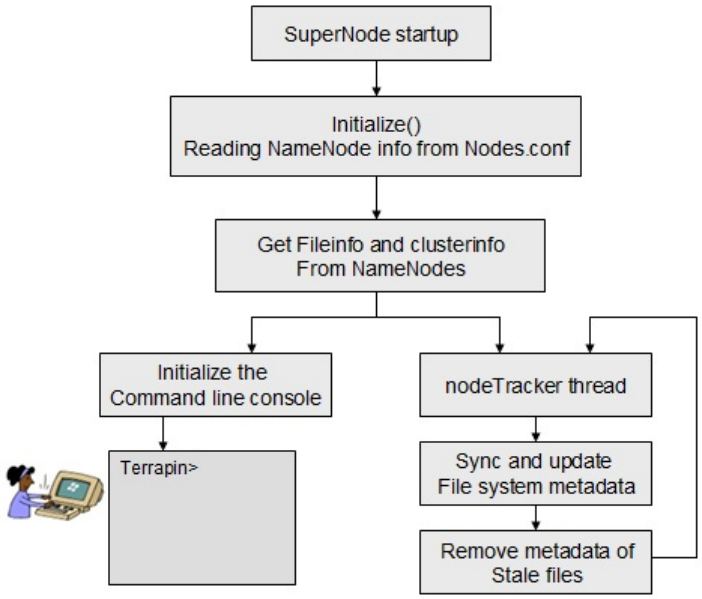


Figure 5.5: SuperNode lifecycle

The SuperNode boots up by reading the node.conf file for available NameNodes. It updates the namenodeURL->NameNode table with all active NameNodes in the configuration. The SuperNode then gathers the file system metadata that is required for startup. It also starts nodeTracker and console threads to finish the boot up.

### 5.3.2 Console

The console on the SuperNode provides users a way to submit commands to the cluster. A user can access data on the cluster, put/get files and submit MapReduce jobs through the command line.

The console is implemented as a thread which runs separately from all the other SuperNode operations. The console continuously receives commands from HadoopT clients

and filters them to the main SuperNode method. The commands are parsed and restructured for the NameNode layer execution. The output of the execution is sent back to the console.

The SuperNode allows most of the common Hadoop commands. Some command line parameters are explained as follows:

dfs $-<config>$

| | |
|---|---|
| $mkdir$ | Creates a folder on the DFS. |
| $ls$ | Lists files on the DFS. |
| $get$ | Gets a file from the DFS to the local path. |
| $put$ | Puts a file into DFS from the local path. |
| $rmr$ | Removes the file or directory from the DFS. |
| $cat$ | Displays file contents on the console. |
| $tail$ | Displays last 1KB of data from the file. |
| $help$ | Help with the command line. |

sfs $-<config>$

| | |
|---|---|
| $add$ | Dynamically add a new NameNode. |
| $remove$ | remove a NameNode. |
| $report$ | retrieve cluster report. |

**MapReduce Jobs**

A user can submit MapReduce jobs through the console. The steps to write MapReduce jobs and job submissions are similar to Hadoop.

*hadoop jar* $<jarpath><classname><input><output>$

# Chapter 6

# Performance Evaluation

The performance of the HadoopT design was measured using actual cluster implementation on commodity hardware. Since we could not demonstrate an actual large scale cluster of commodity hardware because of unavailability of massive real time data, we performed our analysis based on the results we gathered by scaling down the architecture. The experimental environment was created by using small scale datasets and by restricting various parameters of the cluster setup. The parameters that were truncated for this experiment include heap size, data block size and DataNode population in the cluster. Available memory and storage capacity of the cluster were also scaled down for producing extreme conditions of data storage.

## 6.1 Cluster Implementation

For the performance analysis, a prototype of the HadoopT cluster was built on commodity hardware. Two SuperNodes, three NameNodes and nine DataNodes were used for this evaluation. The second SuperNode was introduced to run an additional instance for metadata replication.

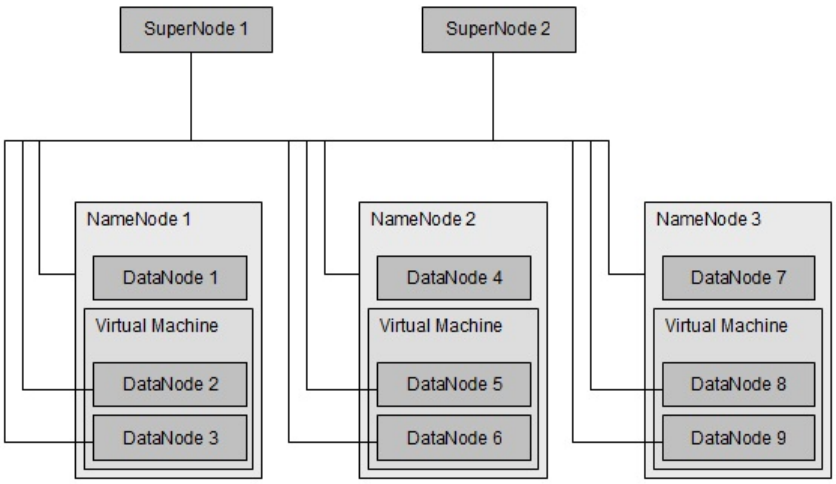


Figure 6.1: HadoopT cluster implementation with two SuperNodes communicating with 3 Hadoop NameNodes and 3 DataNodes each

### 6.1.1 Hardware

For the HadoopT prototype, we constructed the cluster using commodity hardware available at low cost. The machines used were all 2.8 Ghz dual core AMD processor Sun Ultra20s, each with a memory capacity of 1GB. They were equipped with full-duplex 10/100 Mbps ethernet network interface cards connected to a 100 Mbps 3com hub. The hard disk drives were of storage capacity 250 GB at 7200 rpm.

### 6.1.2 Software

Each machine was installed with Ubuntu 10.04 operating system. In order to instantiate number of datanodes in the cluster, VirtualBox 3.2 was installed on all the systems with shared network bandwidth and processing power.

Sun Java 1.6 and Hadoop 0.20 were configured on the second layer NameNode and DataNodes of the HadoopT cluter. All the machines were configured with OpenSSH server to have a password-less access within the cluster. XML-RPC version 3.1.3 library was installed on the cluster to support the compilation and execution process of HadoopT.

## 6.2 Dataset

To evaluate the performance of both the architectures, a Hadoop cluster with the NameNode of heap size 1GB and 3 DataNodes was compared with a HadoopT cluster with a single SuperNode, 3 NameNodes and 9 DataNodes. Upper bound of the data storage capacity was limited to the namespace storage capacity of the NameNode server.

The sample dataset consists of 2 types of experimental data. The text content was taken from gutenberg.org [17] and replicated to generate the experimental set of files. The data set was replicated to the size of 40GB.

1. Dataset consists of files of size 1GB.
2. Dataset consists of files of size 1MB.

Both datasets were used for evaluation and comparison of Hadoop and HadoopT clusters. The environment parameters were regularly modified in order to demonstrate the real time cluster use.

## 6.3 Results

### 6.3.1 Storage Capacity

The storage capacity of HadoopT was measured with the actual data in the cluster. The growth of the namespace was analyzed with respect to increasing amounts of data uploaded to the cluster. The data block size was truncated to 1MB in order to generate more data blocks in the file system. The increase in namespace size is shown in Table 6.1 for the Hadoop cluster and Table 6.2 for the HadoopT cluster with 1 GB file upload on every attempt.

Table 6.1: Incremental Data storage vs namespace size for the Hadoop cluster

| Data(GB) | namespace(MB) | Data(GB) | namespace(MB) | Data(GB) | namespace(MB) |
|---|---|---|---|---|---|
| 1 | 1.84 | 11 | 7.21 | 21 | 12.37 |
| 2 | 2.36 | 12 | 7.72 | 22 | 12.88 |
| 3 | 2.88 | 13 | 8.23 | 23 | 13.39 |
| 4 | 3.4 | 14 | 8.74 | 24 | 13.90 |
| 5 | 3.91 | 15 | 9.26 | 25 | 14.42 |
| 6 | 4.43 | 16 | 9.77 | 26 | 14.93 |
| 7 | 4.95 | 17 | 10.29 | 27 | 15.49 |
| 8 | 5.46 | 18 | 10.80 | 28 | 16.00 |
| 9 | 5.99 | 19 | 11.31 | 29 | 16.51 |
| 10 | 6.5 | 20 | 11.83 | 30 | 17.03 |

Table 6.2: Incremental Data storage vs namespace size for the HadoopT cluster

| Data(GB) | namespace(MB) |
|---|---|
| 3 | 5.54 |
| 6 | 7.09 |
| 9 | 8.64 |
| 12 | 10.16 |
| 15 | 11.75 |
| 18 | 13.30 |
| 21 | 14.85 |
| 24 | 16.40 |
| 27 | 17.96 |
| 30 | 19.51 |

**Discussion**

Our results show a linear increase in the namespace size of both the clusters. For analyzing the storage capacities, it is important to understand how the linear growth can affect the data storage and scalability of the architecture.

For Hadoop, the namespace growth is likely to reach the memory upper bound of the NameNode server with the increasing amount of data. For a rapidly growing Hadoop cluster, the scalability is provided only up to the amount of available memory on the centralized namespace server regardless of the number of DataNodes in the cluster.

With HadoopT, use of multiple NameNodes distributes the namespace storage of the

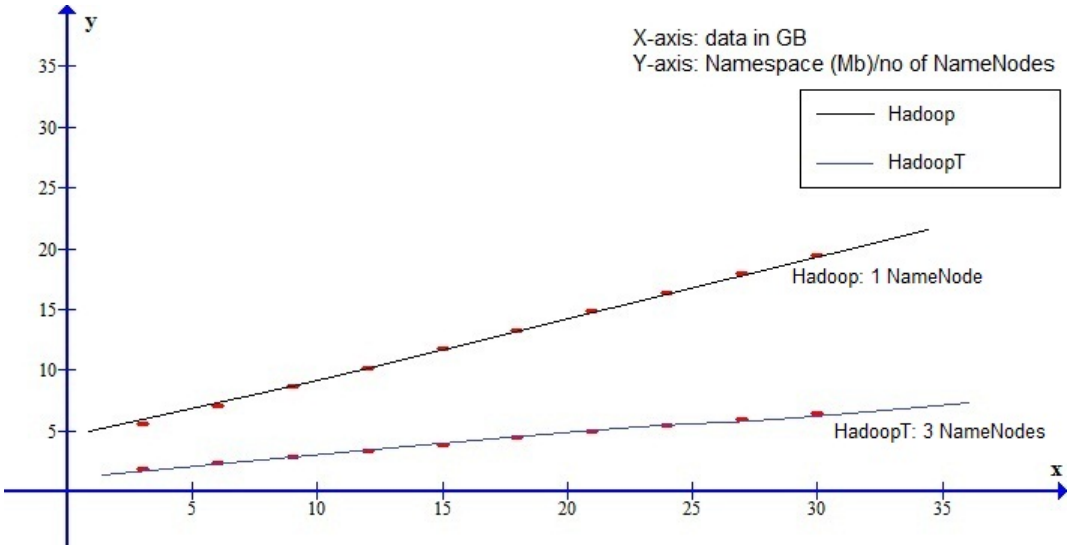


Figure 6.2: Linear growth of data upload for Hadoop and HadoopT

cluster. This reduces the namespace growth rate per NameNode for the cluster resulting to its increased storage capacity. With more number of NameNodes introduced to the cluster, the namespace growth rate is likely to be reduced to provide extended data storage capacity for the client applications.

**Scalability Limitations**

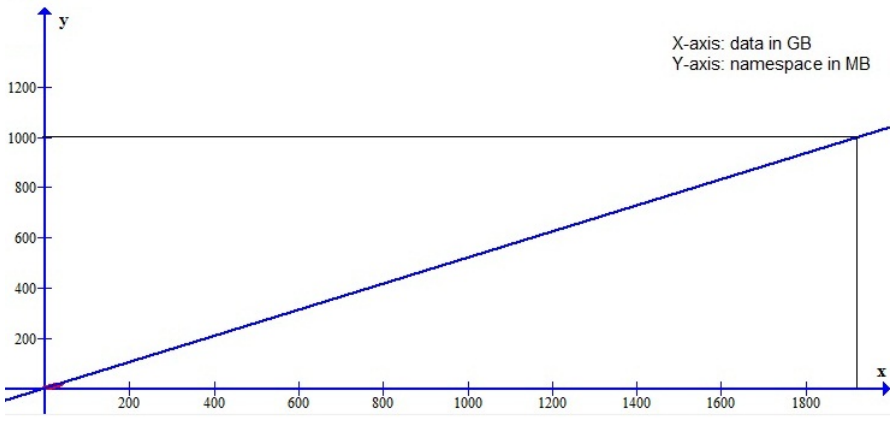


Figure 6.3: Predicted storage capacity of the Hadoop cluster

Figure 6.3 shows predicted growth for the Hadoop cluster with the increasing amounts

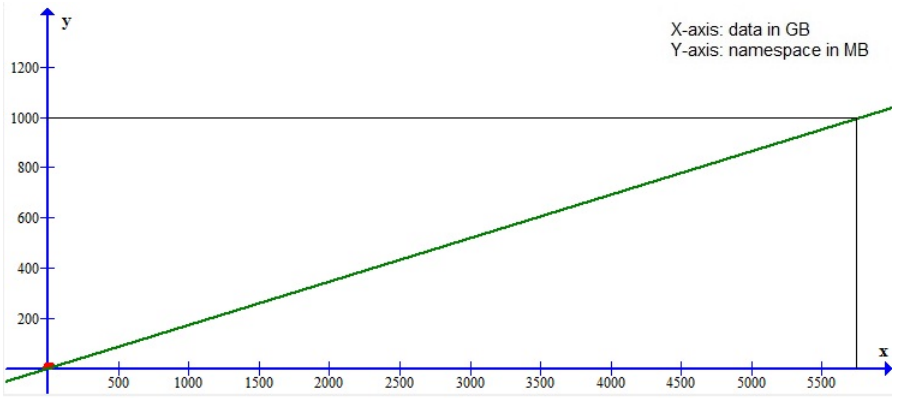


Figure 6.4: Predicted storage capacity of the HadoopT cluster

of data upload. For the Hadoop cluster, with the use of single namespace server, the storage cap is likely to be reached around 1.8PB.

HadoopT cluster with 3 NameNodes for the metadata storage has a lower namespace growth rate per NameNode than the Hadoop implementation. Figure 6.4 shows the storage cap of the HadoopT cluster. With 1GB storage limit of the Hadoop namespace on each NameNode, its data storage capacity is estimated to be approximately 5.4 TB.

**Analysis**

Even though the namespace growth is not an adequate measure of the storage capacities, we can certainly predict the scalability based on the results. HadoopT achieves much higher storage capacity than the default configuration of Hadoop due to its distributed namespace storage.

Konstantin Shvachko [5] estimates that an average file on Hadoop consists of 1.5 blocks with 600 bytes of metadata. With this estimation, it takes 60 petabytes of storage with 60GB memory on the NameNode to store upto 100 million average files.

In comparison, HadoopT takes 50 bytes of memory on the SuperNode irrespective of the block storage. With this estimation, 100 million average files can be stored on HadoopT with 5+GB of memory on the SuperNode and 20+ NameNodes with 3+GB of memory.

If we set the maximum utilization of memory space for a single server without degrading of its performance as 60GB, we can build a HadoopT cluster of 720 petabyte storage. Such a cluster can be built with the help of a single SuperNode server and 12 NameNode servers with 60GB memory each. This storage capacity is 12 times more than the storage capacity of a Hadoop cluster with the same machine configuration.

### 6.3.2 Data Access

Data Read/write commands were executed on the cluster with the help of get/put command line parameters. The data exchange results were taken by uploading the fixed size data to the cluster.

For writing files into the cluster, the SuperNode first moves necessary files to the suitable NameNode. The NameNode then splits the file into fixed size data chunks to store on the DataNodes. For accessing files stored on the cluster, the SuperNode sends a request to the specific NameNode. The SuperNode then moves files from the NameNode and transfers them to the client.

For HadoopT, read/write access rate was observed to be around 5MB/sec. For the Hadoop cluster, read/write access rate was around 7MB/sec.

#### Discussion

For the data read/write operations in Hadoop, data was local to the NameNode. For HadoopT, the data transfer rate was highly affected by the time taken for data transfer between the SuperNode and the NameNode. In a real world scenario, Hadoop users do not have direct access to the NameNode and have to upload data through a submitter node present in the cluster. This extra step should compensate the difference between data transfer rates of both clusters.

### 6.3.3 MapReduce Job Execution

MapReduce job execution shows competitive results for both the architectures. A simple word count job was executed on the files of different size. A word count job counts the frequency of words in the input files. The result is stored as a file and can be accessed using Hadoop commands.

In Hadoop, the NameNode submits MapReduce jobs to the cluster for execution. In HadoopT, the SuperNode forwards MapReduce jobs to specific NameNodes after mapping the input file path. Due to an additional step of forwarding the job, execution of MapReduce jobs in HadoopT is to some extent slower than execution in Hadoop.

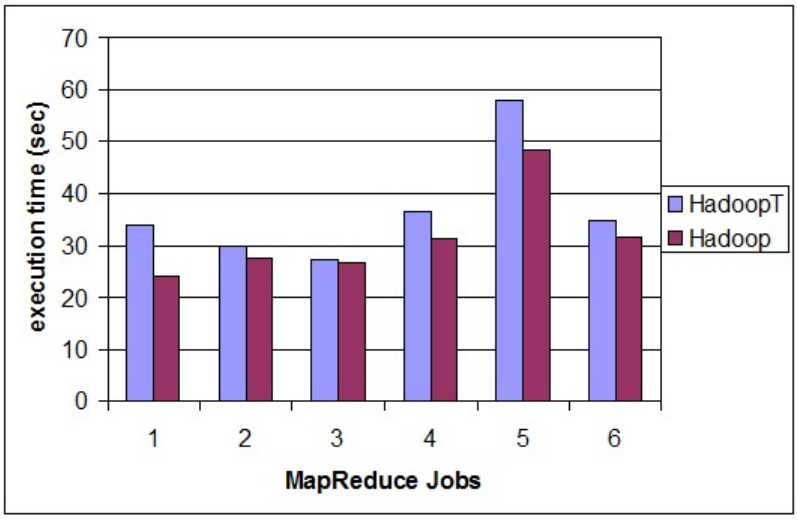


Figure 6.5: Comparison of Mapreduce job execution time in both architectures

Figure 6.5 shows competitive execution times for the word count job executed on the cluster.

#### Discussion

Competitive execution times of MapReduce suggest that the HadoopT cluster is equally efficient as the default Hadoop configuration. However, since only the specific NameNodes participates in the job execution, the remaining cluster is available for other client

applications.

### 6.3.4 Availability

Availability of the cluster was tested by measuring start up times for the SuperNode. In Hadoop, boot up time for the NameNode increases steadily with increasing namespace size. For HadoopT, the SuperNode being the lightweight part of the architecture had a faster boot up time.

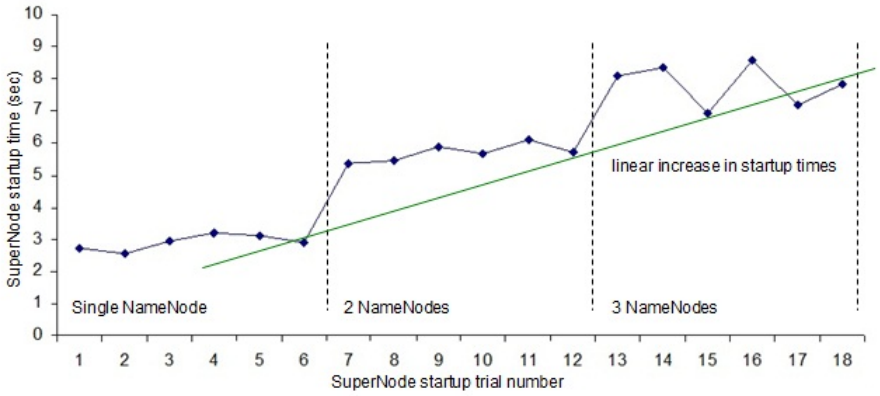


Figure 6.6: Startup time of the SuperNode

Figure 6.6 shows the startup time of the SuperNode for increasing number of NameNodes. This shows the linear increase in the startup times with increase in the number of NameNodes in the cluster. The SuperNode initialization averages to 3.1 seconds per NameNode configured in the cluster.

#### Discussion

In Hadoop, when the NameNode fails, the entire cluster goes offline for the entire downtime period. Even though Hadoop has a backup secondary NameNode for the namespace, it is not an active node for client access. For HadoopT, two or more instances of SuperNode can run simultaneously in the cluster. Due to multiple instances of the SuperNode, its downtime does not prevent users from accessing the cluster. However, the overall startup time of the

SuperNode linearly increases with number of NameNodes configured in node.conf.

Occurrence of failure of the second layer of HadoopT leads to partial failure of the cluster during downtime of the NameNode. However, during this period, a user is still able to access the residual portion of the cluster through the SuperNode.

The result shows that HadoopT presents a fault tolerant architecture with multiple instances of metadata servers. Multiple SuperNodes make the system highly available.

# Chapter 7

# Conclusions

This thesis was initially motivated by debate in the Hadoop community and computer industry regarding the centralized namespace storage and single point of failure. Hadoop Distributed File System is being widely used by many organizations for its scalability, reliability and lower cost of implementation. But Hadoop has been called into question recently, as the research [5] outlined several limitations imposed on the architecture because of its centralized namespace storage.

This work took an in depth look into the architecture of Hadoop and its implementation details. The intent was to identify and analyze the drawbacks of the Hadoop file system. We discussed several disadvantages which were likely to be caused due to the centralized namespace server. Related work from the previously published papers and articles was discussed and the detailed analysis of solutions for the limitations was presented, which provided the foundation for our research.

We proposed the design of a distributed metadata storage architecture, HadoopT, for a Hadoop file system involving multiple NameNodes. The focus of our work was to distribute the file system metadata across multiple namespace servers, thus eliminating the single point of failure. The HadoopT architecture was implemented not only with the intent of achieving improved scalability and high availability, but also for accommodating several functionalities provided by Hadoop.

Finally, we evaluated the architecture considering several key aspects of the file system. For this work, we built a cluster using commodity hardware and implemented HadoopT and the default architecture of Hadoop. The linear scalability of file systems was demonstrated by incrementally uploading of the data. The evaluation was performed as a comparison between both architectures on the basis of storage capacities, namespace storage growth per NameNode, data exchange rates and MapReduce job executions. Our design of multiple NameNodes increases both storage capacity and availability of the cluster by eliminating the single point of failure.

## 7.1 Future Work

This work implemented a distributed namespace storage architecture for Hadoop using multiple NameNodes. Several tasks of the system can be parallelized in order to improve the performance of the system. The periodic node tracker can be invoked simultaneously for all the NameNodes to give better and faster refreshing rate of the file system metadata.

In the future, boot up time of the SuperNode can be reduced with the help of parallel initialization. Further, data transfer rates between various nodes can be improved for faster and better performance. Finally, the performance evaluation of this architecture can be done with the help of several existing Hadoop client applications.

More advanced technique for metadata storage could be designed and investigated for the cluster storage. The metadata storage could be improved for further scaling of the file system. The resulting distributed namespace architecture may change the development directions for future large scale data storage systems.

# Bibliography

[1] Michael Armbrust, *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," UC Berkeley Reliable Adaptive Distributed Systems Laboratory, Feb. 2009.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System," Google, 2003.

[3] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Google, 2004.

[4] Konstantin Shvachko, *et al.*, "The Hadoop Distributed File System," Mass Storage Systems and Technologies (MSST), IEEE 26th Symposium on IEEE, 2010, http://storageconference.org/2010/Papers/MSST/Shvachko.pdf.

[5] Konstantin V. Shvachko, "HDFS scalability: the limits to growth," usenix vol 35 no 3, May 2010, www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf.

[6] "HDFS scalability with multiple NameNodes," Hadoop Jira on Apache, Sept 2010, https://issues.apache.org/jira/browse/HDFS-1052.

[7] Dhruba Borthakur, "The Hadoop Distributed File System: Architecture and Design," Nov 2007.

[8] Hadoop wiki page documentation, http://wiki.apache.org/hadoop/.

[9] Hadoop Java API, Sept 2010, http://lucene.apache.org/hadoop/api/.

[10] Dhruba Borthapur, "The Curse of the Singletons! The Vertical Scalability of Hadoop NameNode," facebook, Apr 2010, http://hadoopblog.blogspot.com/2010/04/curse-of-singletons-vertical.html.

[11] George Porter, "Decoupling Storage and Computation in Hadoop with SuperDataNodes," ACM SIGOPS Operating Systems Review, Volume 44 issue 2 Apr 2010.

[12] Derek Tankel, "scalability of Hadoop Distributed File system," Yahoo developer work, May 2010.

[13] The RPC server Listener thread is a scalability bottleneck, Apache Jira, https://issues.apache.org/jira/browse/HADOOP-6713.

[14] Readers-writer lock, http://en.wikipedia.org/wiki/Readers-writer-lock.

[15] Dhruba Borthapur, "Hadoop AvatarNode High Availability," Facebook, http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html.

[16] Feng Wang, *et al.*,, "Hadoop High Availability through Metadata Replication," IBM China Research Laboratory, ACM, 2 Nov 2009.

[17] Project gutenberg, http://www.gutenberg.org/wiki/Main-Page.

[18] Hadoop powered by http://wiki.apache.org/hadoop/PoweredBy.

[19] Baidu, web search engine http://www.baidu.com/.

[20] Facebook, Social networking website http://www.facebook.com/.

[21] Dhruba Borthakur, Hadoop blog, http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html.

# Appendix A

# Classes

## A.1 HadoopTServer

It is an XML-RPC server component of the architecture. It starts the web and RPC server to establish the communication protocol between the NameNode and the SuperNode. The HadoopTServer also holds a local copy of the fileinfo data structure to keep in sync with the SuperNode.

```
public static Hashtable<String, String>
fileInfo = new Hashtable<String, String>();

//Starting the web and RPC server
WebServer server = new WebServer(6666);
XmlRpcServer xmlRpcServer = server.getXmlRpcServer();

PropertyHandlerMapping phm = new PropertyHandlerMapping();
//Adding supernodeProtocol handler
phm.addHandler("SupernodeProtocol", SupernodeProtocol.class);

xmlRpcServer.setHandlerMapping(phm);
XmlRpcServerConfigImpl
serverConfig = (XmlRpcServerConfigImpl)xmlRpcServer.getConfig();
```

```
serverConfig.setEnabledForExceptions(true);
serverConfig.setContentLengthOptional(false);

server.start();
```

Figure A.1: XMLRPC server implementation

## A.2 supernodeProtocol

The supernodeProtocol class consists of functions with which the SuperNode communicates with the NameNode. A user can manipulate the low directory namespace on the NameNode as well as perform read/write file stream operation.

```
public class SupernodeProtocol
{
public int sendHeartbeat();
public Hashtable<String, String> getFileInfo(String str)
throws IOException, InterruptedException;
public HashMap<String, String> getClusterInfo(String str)
throws IOException, InterruptedException;
public Hashtable<String, String> executeCommand(String cmd)
throws InterruptedException, IOException
}
```

Figure A.2: SuperNodeProtocol class

getFileInfo(String str): generates the file system information and parameters and returns them to the SuperNode in the form of a Hashtable.

getClusterInfo(String str): generates the cluster information such as storage size, empty space and number of DataNodes in the cluster and returns them to the SuperNode in the

form of a Hashtable.

executeCommand(String cmd): Receives Hadoop commands from the SuperNode and executes them on the NameNode. The result is gathered in the form of a Hashtable and sent back to the SuperNode.

## A.3 NameNode

NameNode is a XML-RPC client object for the HadoopT server. It connects with the HadoopT server with the help of supernodeProtocol class implemented on HadoopT server. The NameNode object is invoked by the SuperNode to execute commands on the cluster.

The NameNode class implements SuperNode side methods for supernodeProtocol. These methods are wrapper functions for the RPC calls made by the SuperNode. The NameNode checks the input commands submitted on the SuperNode for their validity and parses them for execution on the HadoopT server.

```
public Namenode(String nodeIP, String nodePort) {
namenodeIP = nodeIP;
namenodePort = nodePort;
conf = new XmlRpcClientConfigImpl();
try {
     //connect to the namenode RPC servers
conf.setServerURL(new URL("http://" + nodeIP + ":" + nodePort));
conf.setEnabledForExtensions(true);
conf.setConnectionTimeout(60 * 1000);
conf.setReplyTimeout(60 * 1000);

    node = new XmlRpcClient();
    node.setConfig(conf);
}
```

```
catch(MalformedURLException e)
{ e.printStackTrace(); }
}
public int sendHeartbeat();
public boolean getFileInfo(String param)
throws IOException, InterruptedException;
public boolean getClusterInfo()
throws IOException, InterruptedException;
public boolean executeCommand(String command)
throws IOException, InterruptedException {
HashMap<String, String> result = new HashMap<String, String>();
try {
//Create the parameters we want to pass to the server
     Object[] params = new Object[]{new String(command)};
// Execute the command, send the data to the server
    // (Class Output, method outputString)
result = (HashMap<String, String>)
node.execute("SupernodeProtocol.executeCommand", params);

for (int i = 0; i < result.size(); i++)
System.out.println((String)result.get(Integer.toString(i+1)));
}
catch (XmlRpcException e)
{ e.printStackTrace; }
return true;
}
```

Figure A.3: NameNode

## A.4 SuperNode

The SuperNode serves both as a high level directory manager and clients access point for the HadoopT file system. It manages three critical data structures; namenodeURL->namenode, fileinfo->namenode and clusterinfo->namenode. Single or multiple instances

of the SuperNode can run in any HadoopT deployment. The SuperNode exposes the distributed file system to the outside world for access, storage and job execution.

```
SuperNode supernode = new SuperNode();
Scanner nodesc;
nodesc = new Scanner(new File("nodes.conf"));
while(nodesc.hasNextLine()) {
String nodedetails = nodesc.nextLine();
String [] nd = nodedetails.split(":");

Namenode nnode = new Namenode(nd[0], nd[1]);
supernode.namenode.put(nd[0], nnode);
}
//Initializing the supernode
supernode.Initialize();
//THREAD: To keep the file system metadata updated
NodeTracker ndTracker = new NodeTracker(supernode);
ndTracker.start();
```

Figure A.4: SuperNode initialization

The SuperNode initializes itself by reading the configuration file to learn the available NameNodes in the cluster. It invokes the getfileinfo and getclusterinfo methods to receive all the necessary metadata required to manage the cluster.

The SuperNode console gives a user means to execute HadoopT command lines. It communicates with the NameNode layer by invoking a supernodeProtocol object. The SuperNode also runs a tracker to check all the live NameNodes and to keep its namespace updated. The tracker methods are repeatedly invoked by the SuperNode for all the NameNode objects.

## A.5 nodeTracker

The nodeTracker is a separate thread that runs on the SuperNode. It keeps track of all the NameNodes in the cluster using the namenodeURL->namenode data structure. The nodeTracker thread repeatedly invokes methods on the HadoopT server to keep the SuperNode metadata up-to-date.

```
public void run() {
while(true) {
// set all the files to false mode
// Poll each namenode for modified metadata
// remove the stale copies from the supernode metadata
this.removeStale();
}
}
```

Figure A.5: nodeTracker implementation

## A.6 dfsFunctions

dfsFunctions class implements all the file system command executions on the SuperNodes. A user can execute generic Hadoop commands on the SuperNode to access file system information or execute MapReduce jobs. The commands for file system operations such as list, get, put, create, delete, cat, tail work with the help of this class. The dfsFunctions object is invoked by the SuperNode and is executed by the console thread everytime a command is accepted.

```
public class dfsFunctions {
```

```
// prints all the necessary the file information: ls
public void printFileData(SuperNode supernode, String cmd);
// prints cluster report: -report
public void printReport(SuperNode supernode);
// creates folder on the FS.: -mkdir
public boolean makeDirectory(SuperNode supernode, String cmd);
// put file on the cluster: -put
public boolean putData(SuperNode supernode, String cmd);
// get file from the cluster: -get
public boolean getData(SuperNode supernode, String cmd);
// read file contents: -cat tail
// tail command prints last 1kb data from the file
public boolean catOrTailFile(SuperNode supernode, String cmd);
// remove file from the cluster: -rm
public boolean removeFileInfoEntry(SuperNode supernode, String cmd);
// remove all files from the specific folder: -rmr
private boolean removeAllElements(SuperNode supernode, String nodefile);
// return namenode url for the file path
private String getpathCluster(SuperNode supernode, String dirpath);
// returns free namenode on the cluster
public String getFreeCluster(SuperNode supernode);
}
```

Figure A.6: file system functions

# Appendix B

# HadoopT Installation

Steps to install a HadoopT cluster:

1. Install Ubuntu 10.04 with Java 1.6 on the machines.
2. Install one or more Hadoop clusters as per the requirement with single NameNode and multiple DataNodes. (following the Hadoop installation guideline)
3. On each cluster, install the HadoopT server.
4. Startup the Hadoop Distributed File System and the HadoopT server
5. For Starting the SuperNode, configure the node.conf file with each necessary NameNode.
   node.conf can be configured as a pair of NameNode and HadoopT port per line.
   node.conf:
   $< nodeip >< portnumber >$
   ...
6. Boot up the SuperNode to gain access to the console for the HadoopT cluster.