An Approach to Geographic Based Namespace Load Distribution using Symphony

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By
Ravneet Kaur
(Roll No. 801132025)

Under the supervision of:
Dr. Shalini Batra
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004
July 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*An Approach to Geographic based Namespace Load Distribution using Symphony*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Shalini Batra* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Ravneet Kaur

Signature:

(Ravneet Kaur)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Shalini Batra)
Assistant Professor,
Computer Science and Engineering Department,
Thapar University, Patiala

Countersigned by

Dr. Maninder Singh
Head
Computer Science and Engineering Department
Thapar University
Patiala

Dr. S. K. Mohapatra
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

I express my sincere gratitude to Dr. Shalini Batra, Assistant Professor, Department of Computer Science and Engineering, for her inspiration, guidance, comments and suggestions during the research work. Her continuous supervision and support helped me to keep this dissertation work on the right track and achieve this final research thesis. Her ingenuity and kindness has motivated me along the way till the completion of this work.

I am also thankful to Dr. Maninder Singh, Head of Department of Computer Science Engineering, for his valuable suggestions.

Many thanks to my family and friends for many rewarding discussions and for providing help when ideas and questions were needed to be discussed.

Finally, I express my earnest thanks to my parents and Almighty who helped me at every step, showing me the right direction and to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Ravneet Kaur

Ravneet Kaur

(801132025)

Abstract

With the enormous growth of volume of data, Namespace distribution and replication are becoming a major challenge as Namenode server needs to coordinate with data nodes for its status, job execution and data blocks operations. As the data is growing day by day, Hadoop Usage of Namespace impose big challenge to achieve high scalability, availability and single point of failure.

The proposed approach of hadoop is based on Symphony and has resolved the issues of namespace scalability, availability and address the issue of single point of failure. The focus of work is based on geographic based namespace distribution using Symphony. By distributing the metadata storage, the Hadoop distributed file system has gain immense storage capacity by eliminating the bottleneck of the single NameNode. The approach provides the ability to dynamically add Namenodes as required, thereby increasing the underlying performance and scalability. The multiple NameNode approach contributes to distribute the workload of the Hadoop file system and eliminate the single point of failure. A comparison of Chord and Symphony has been provided and it has been analyzed that Symphony works better for geographic based key distribution technique on Namespace server.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The exponential growth of internet based applications like Google, Facebook, Yahoo, Youtube, twitter, eBay, IBM, Amazon, *etc.* has led to vast amount of information leading to changes in the techniques to store and analyze voluminous data. Since the enterprises are facing problem to store and process large amount of data produced by these applications they are looking for solutions which are reliable to store and process big data. Since organizations need to search and index huge volume of data various data intensive computing models are evolving to cater such future needs. A variety of system architectures have been implemented for data-intensive computing and large-scale data analysis applications including parallel and distributed relational database management systems.

Parallel processing approaches can be generally classified as either compute-intensive or data-intensive. Compute-intensive is used to describe application programs that are compute bound. Such applications devote most of their execution time to computational requirements as opposed to input-output and typically require small volumes of data. Data-Intensive Computing is a class of parallel computing applications which use a data parallel approach to process large volume of data typically terabytes or petabytes in size and typically referred to as Big Data.

Computer system architectures which can support data parallel applications are a potential solution to the terabyte and petabytes of data. Data-Intensive Computing Systems should have approaches to parallel programming to address the parallel processing of data on data-intensive systems, design of data-intensive computing platforms to provide high levels of reliability, efficiency, availability, and scalability and lastly identifying applications that can exploit this computing paradigm. Hadoop is one such architecture which exploits above said features.

1.1 Hadoop Architecture

Hadoop, inspired by Google's File System (GFS) [1], is a flexible and easily available architecture for data intensive computing for Large Scale Computation and Data Processing [2]. It does not require expensive, highly reliable hardware to run on. It is designed to run on clusters of commodity hardware (commonly available hardware). Nodes are commodity PCs with Gigabit Links with the rack switches and rack switches are connected to each other with 3-4 Gigabit Connectivity.

Hadoop parallelizes data processing across many nodes (computers) in a cluster. It speeds up large computations and hides I/O latency. It is especially well-suited to large data processing tasks like searching and indexing because it has powerful distributed file System [2, 3, 4]. The Hadoop distributed file system has namenode servers and data nodes. The namenode server maintains the metadata called namespace. Namespace has information about namenode servers, file, blocks, replica, data nodes and running jobs. It is highly reliable as it replicates chunks of data to nodes in the cluster. The replica decisions are used to improve the availability of system.

Hadoop is used for various applications such as searching, log processing, business intelligence, data warehousing, video and image processing, sound processing, NLPs and applications in which data is divided into sub data sets. Hadoop core consists of Hadoop Distributed File System (HDFS) and MapReduce.

1.2 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) is a file system designed for storing very large files with streaming data access patterns, running on clusters on commodity hardware [5]. It enables applications to work with thousands of computation-independent computers and petabytes of data. It connects nodes placed in clusters over which data files are distributed. Access to data files is handled in a streaming manner, i.e., applications or commands are executed directly using the MapReduce processing mode. The namenode splits the large file into fixed size data blocks of 128 MB in size. These blocks are scattered across the cluster. This type of data storage follows the write once read many (WORM) model. The files once written are only appended deleted and cannot be

modified to maintain the data coherency. Since Hadoop is ideal for storing large amount of data, like terabytes and petabytes, and uses HDFS (Hadoop Distributed File system) as its storage system, it is used for distributed and parallel processing of data.

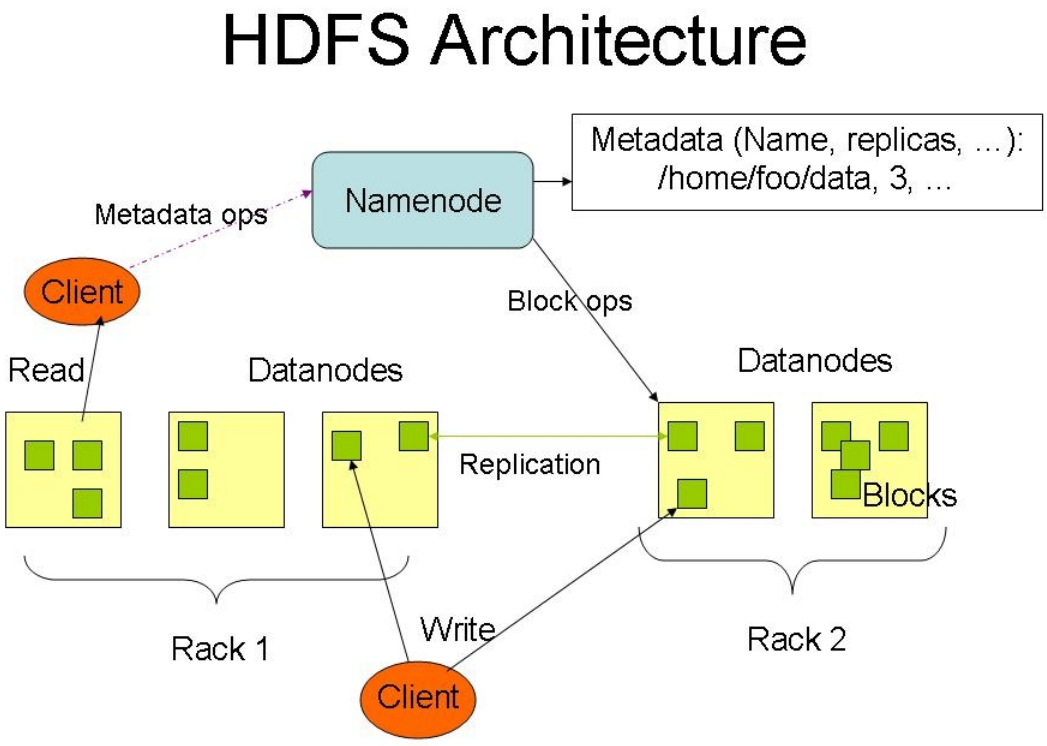

Figure 1.1: HDFS Architecture [5]

1.2.1 Namenodes and Datanodes

HDFS has a master/slave architecture consisting of interconnected clusters of data nodes where files and directories reside. The cluster consists of a single NameNode [3], a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of Data nodes that store data as blocks within files. Internally, a file is split into one or more blocks and these blocks are stored in a set of data nodes. The name node executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to

data nodes. The data nodes are responsible for serving read and write requests from the file system's clients. The data nodes also perform block creation, deletion, and replication upon instruction from the name node.

1.2.2 Namespace

The namespace is a live record of the HDFS located on the centralized name node 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.

1.3 MapReduce

Hadoop implements a computational paradigm named map/reduce[6] which is a programming paradigm where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. MapReduce jobs are submitted to namenode and these are then forwarded to datanodes where the data block resides. Data nodes execute the jobs on local machine and return the results. The computation in MapReduce is divided into two tasks, one is called Mapper and other is called Reducer [6, 7]. In the mapping task, the data is processed into key and the value pairs with a minimal coordination of data nodes. In the reducing task, each output from data nodes is combined to produce single output for the application.

In addition, it provides a distributed file system that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both map/reduce and the distributed file systems are designed to address the data node failures.

1.3.1 Jobtracker and Tasktracker

The job tracker is a daemon that runs on the namenode server and coordinates the execution of jobs running on the data nodes. The task tracker is daemon that runs on data nodes to monitor the tasks running on it. In case of failure, the task tracker sends failure notice to name node and name node initiates same job on replica data node and inform the job tracker to update the status.

Users submit their MapReduce jobs to job tracker. The JobTracker redistributes these jobs to the slave nodes. The JobTracker tries to make sure that it pushes the work to the Task Trackers that are closest to the data of that task to avoid unnecessary IO operations.

1.4 Goals of HDFS [8]

- **Data replication:** HDFS replicates file blocks for fault tolerance. An application can specify the number of replicas of a file at the time it is created, and this number can be changed any time. The name node makes all decisions of block replication in data node.
- **Data organization:** One of the main goals of HDFS is to support large files. The size of a typical HDFS block is 64MB. Therefore, each HDFS file consists of one or more 64MB blocks. HDFS tries to place each block on separate data nodes.
- **Data Locality:** A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system.
- **Commodity hardware:** Hadoop is a flexible and available architecture for large scale computation and data processing on commodity hardware. It does not require expensive and highly reliable hardware. It is designed to run jobs on clusters of commodity hardware i.e. commonly available hardware.
- **Data storage reliability:** One important objective of HDFS is to store data reliably, even when failures occur within name nodes, data nodes or network partitions. Detection is the first step HDFS takes to overcome failures. HDFS uses heartbeat messages. (A heartbeat message is a message sent from an originator to a destination that enables the destination to identify if and when the originator fails or is no longer available) to detect connectivity between name and data nodes.
- **HDFS heartbeats:** Since several things can cause loss of connectivity between name node and data nodes, each data node sends periodic heartbeat messages to its name node, so the latter can detect loss of connectivity if it stops receiving them. The nodes which are not responding are marked as dead data nodes by name node and data stored on a dead node is no longer available to an HDFS client from that node. If the death of a node causes the replication factor of data blocks to drop below their minimum value, the name

node initiates additional replication to bring the replication factor back to a normalized state.

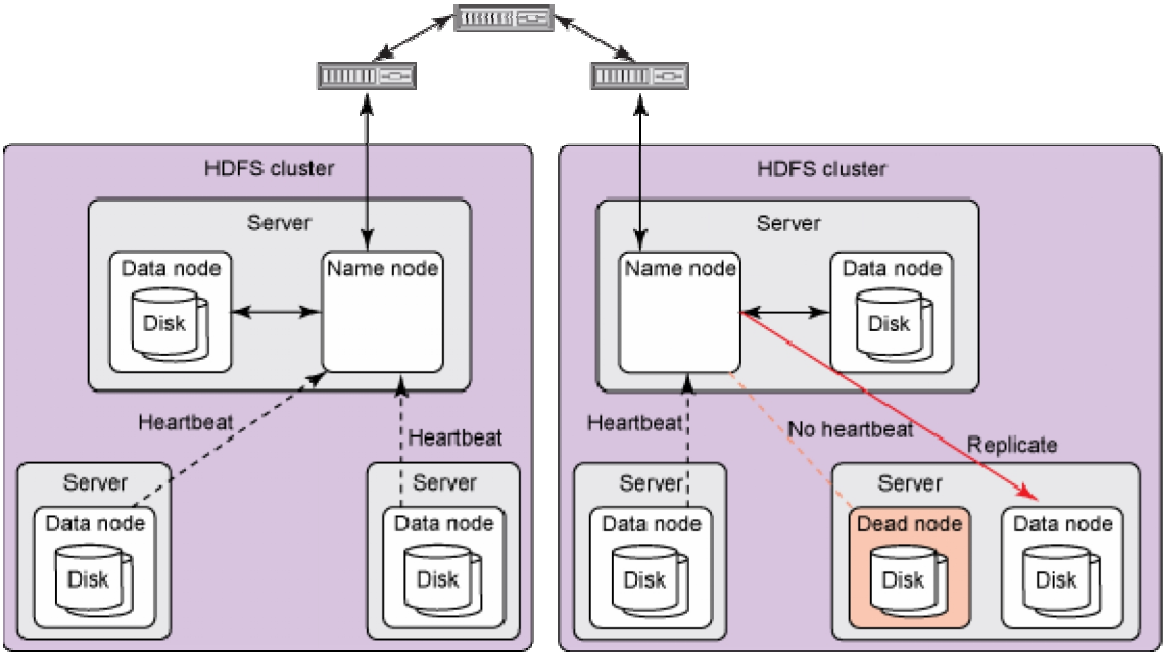

Figure 1.2: HDFS heartbeat Process [8]

- **Block Placement:** When the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a different node in the local rack, and the last on a different node in a different rack. This policy cuts the inter-rack write traffic which generally improves write performance.

1.5 Limitations of HDFS

1.5.1. Scalability Issue

Scalability of the NameNode has been a key issue. Because the NameNode server keeps all the namespace and block locations in memory, the size of the NameNode heap has limited the number of files and also the number of blocks addressable i.e. the number of namespace objects the single centralized namespace server can handle. As the storage capacity of a cluster increases, more memory in namenode server is required [9]. But it is unrealistic to add any amount RAM to the name node server. As the number of data

nodes increases, the work load on a single centralized namenode server increases and has a great impact on the performance and availability of the cluster. It is therefore, increasing work load and memory that restricts the scalability of the Hadoop cluster.

1.5.2 Availability Issue

A single centralized namenode server is more prone to failures. In case namenode server fails, the whole cluster data nodes are unavailable to client applications. The recovery time depends on the amount of metadata data [9]. As the name node server starts, it takes time to load the namespace to cache. This adds up to the unavailability of service. So a single centralized namenode server becomes a Single Point of Failure (SPOF) and this failure may be attributed because of denial of service or distributed denial of service attack.

1.5.3 Performance Issue

Namenode is responsible to check the heartbeat of data nodes, maintaining the namespace of the cluster and responding to client application queries for metadata. As the data nodes and data blocks grow in the cluster, the performance reduces after reaching the thresholds. So a single centralized namenode becomes a bottleneck. The whole cluster performance depends on the performance of name node server.

Despite the popularity of Hadoop, the effectiveness of the HDFS has been questioned in recent research considering its single NameNode architecture [4]. 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 issue and fails to accommodate a significant data growth rate beyond its memory limitations.

1.6 Thesis Organization

The thesis report comprises of six chapters. The first chapter gives introduction to the data intensive computing, Hadoop approach, its components and MapReduce jobs and HDFS goals. In this chapter, the architectural limitations like scalability, availability, performance issues are discussed. Second chapter concentrates on the efforts of

researchers and their recent published approaches to overcome the limitations. A brief discussion of these design and techniques is presented. In third chapter, the problem of statement is stated. In fourth chapter, a new approach to overcome these issues is proposed. In this proposed Hadoop approach, the namespace is distributed by using distributed hash tables. Fifth Chapter presents the performance evaluation of proposed Hadoop with the existing approaches. Last chapter concludes our work and presents the future scope.

Chapter 2

Literature Review

A key challenge faced by enterprises today involves efficiently storing and processing large amount of data as the data is in terabytes or petabytes. 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. The companies are facing problem how to store and process that large amount of data. Yahoo was first who uses its large datasets to support research for advertisement systems. Then the MapReduce programming model [6] 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) [1]. The success of the Google File System and MapReduce inspired the development of Hadoop [2], an open source framework for building large clusters.

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. Hadoop 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.

Hadoop was first started by Doug Cutting to support two of his other well known projects, Lucene and Nutch Hadoop and it was renamed to Hadoop. In 2008 Yahoo, one of the significant driving forces behind Hadoop announced that their web search engine index was being generated by a 10,000 core Hadoop cluster. Doug Cutting, who created

Hadoop, named after his son's stuffed elephant and the only analogy is size of elephant and size of data hadoop process (Bigdata).

Although, Hadoop distribute the data or files on multiple datanodes by spiriting the file into blocks but still has issues with namenode server that need to be address are discussed below:

- Single Namenode server or namespace residing on single namenode server has been susceptible to single point of failure
- Namespace is residing on single namenode server and is cached every time the namenode starts to process client queries more efficiently. In this approach, only vertical scalability is achieved by adding more memory and CPU. There is always an underline limit for both memory and CPU for an Operating system.
- Periodically, Namenode records the heartbeat signal of every single data node and status of each block residing on it. This limits the number of datanodes it can support. Till date, only 4,000 data nodes has been implemented and tests are going on to support 10,000 data nodes [9]. Each datanodes registers itself with namenode. So, data node generates a load that single namenode cannot handle.
- Each namenode maintains job tracker to maintain and monitor the job execution and each data node has task tracker service to monitor the tasks. If a particular task has not been responding the namenode reschedule the task on other node where same block is present. So it requires coordinating with task tracker job. High number jobs puts heavy load on the namenode. So, namenode load should be distributed.
- Due to the load, Client requests are not processed timely and user gets slow response and even outage of service. This situation is unacceptable to any business house.

The study has shown that namespace distribution and replication is the technique to address these open issues. It allows the namenode servers to cater the growing demand of namespace and load associated with it.

2.1 Strategies to Improve Centralized Namespace Server

Since the weakness of the centralized namespace storage of Hadoop has surfaced up, there have been attempts in research publications providing strategies for eliminating the single point of failure and address the scalability issue of the architecture. In this chapter, the techniques to address these issues are discussed.

2.1.1 AvatarNodes: the Active/Standby AvatarNode

Dhruba Borthapur discussed the issue of single point of failure of Hadoop and suggests improvements in failover of Namenode server. The AvtarNode [10] was developed to address the issue of failover and a mechanism to deal with the single point failure of the Namenode. The primary AvtarNode runs exactly same as the Namenode and writes its transaction logs into the shared NFS filer. Another instance of AvtarNode is instantiated that runs in standby mode. Using NFS, it keeps reading the transaction logs from the same shared NFS filer and keeps feeding the transaction logs to the encapsulated Namenode instance. The name node within standby AvtarNode is kept in safe mode to prevent it from participating in the cluster activities. The approach is show in figure 2.1.

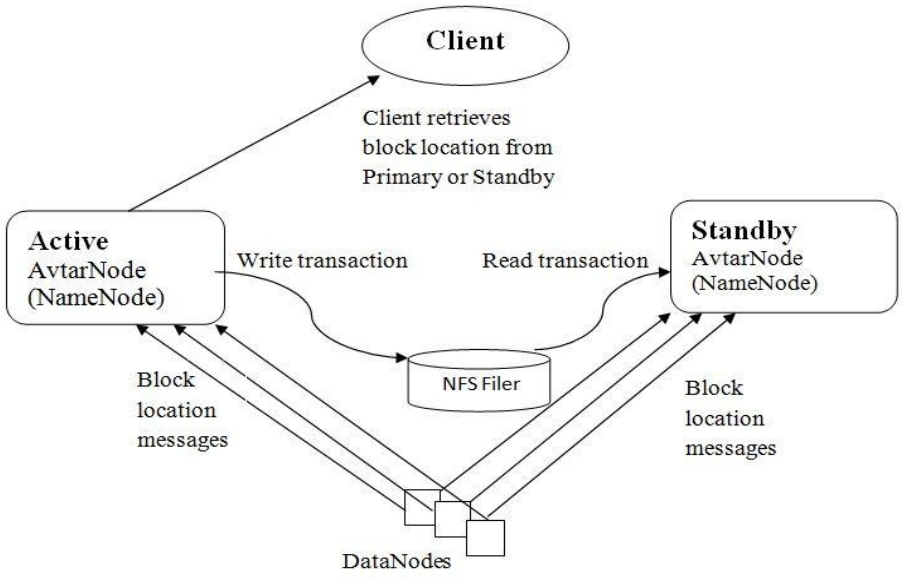

Figure 2.1 AvatarNodes: the Active/Standby AvatarNode [10]

HDFS clients are configured to access the AvatarNode via virtual IP address (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. When 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. Hence the failover happens in minimum time and kept all functionalities intact.

The AvatarNode is effective mechanism to guard against Namenode failures and keeps the namespace data protected. However, the AvatarNode does not address the high scalability of the architecture and still has the single point of access to the cluster. As the namespace grows, the two name node servers do not load balance the work. This approach provides failover and nor able to accommodate the large namespace.

2.1.2 Replication NameSpace Technique for single point of failure

Feng Wang's [11] discussed the metadata replication based solution to provide high availability and failover technique. 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. 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 Edit Log transactions are stored as a backup copy on the Namenode, the solution provided in this paper only concentrates on the replication of critical metadata.

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 2.2 explains the architecture designed for critical metadata replication. The primary node collects metadata from clients request processing threads and sends it to the slave node. The slave node handles the processing of the received 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 takes

place of the primary node. Upon selection of a slave node as the primary node, it changes its IP address to the IP address of the failed primary node, thus finishing the failover

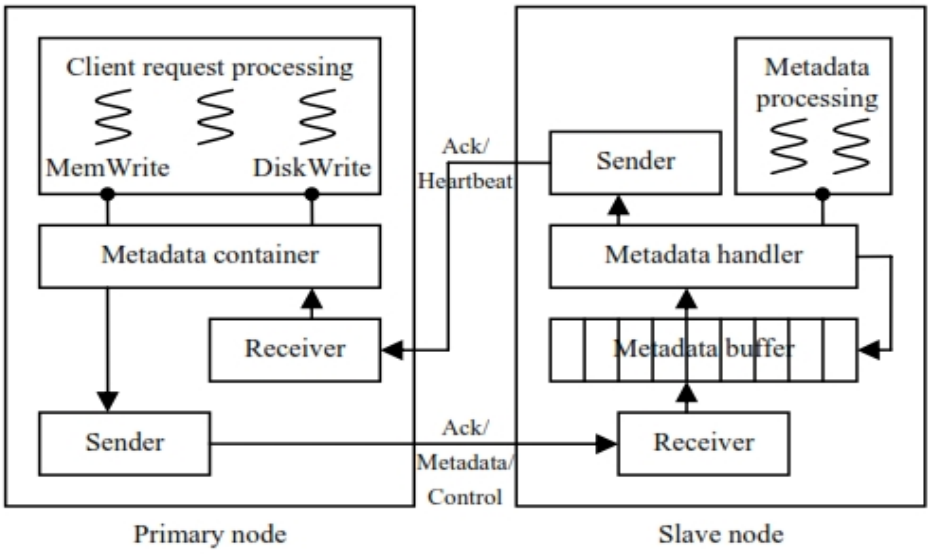

Figure 2.2 Replication of Namespace [11]

It presents an adaptive method for failure recovery of the Namenode by metadata replication with further reduces 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. The metadata replication does not improve the scalability of the architecture.

2.1.3 Architecture of SuperDataNode

George Porter [12] provides a solution to meet the increasing demands of namespace storage of the cluster. As shown in Figure 2.3, Porter [12] discusses the use of a decoupled Datanode architecture to provide increased data storage and computation in Hadoop. In this paper, SuperDataNodes is introduced 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.

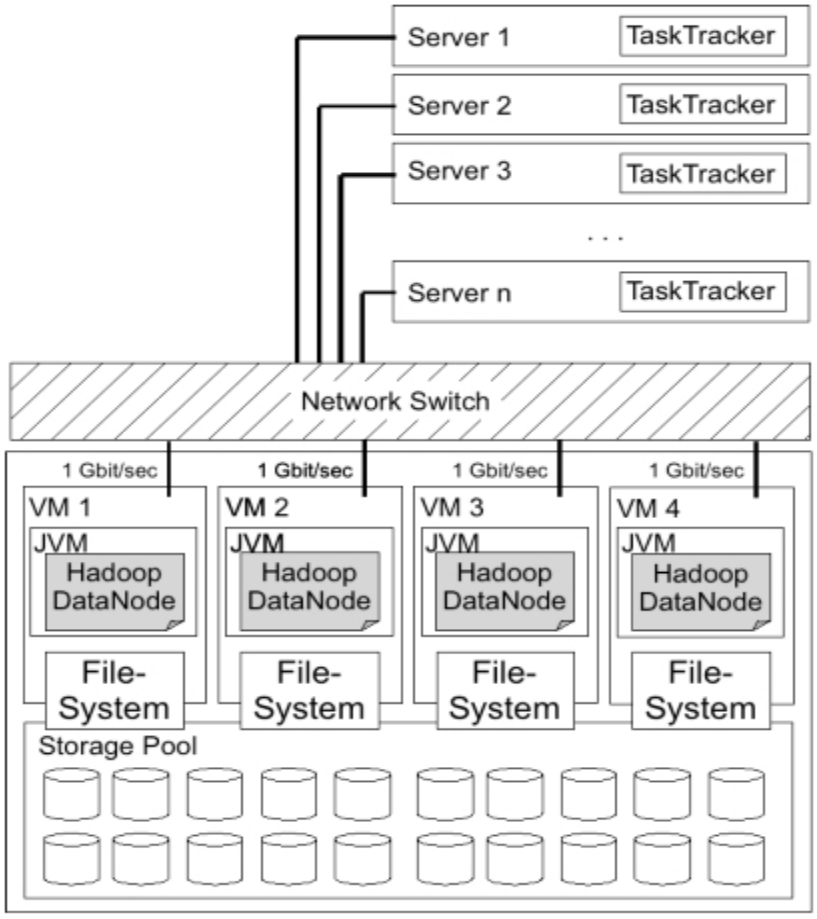

Figure 2.3: Architecture of SuperDataNode [12]

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.

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 metadata 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.

2.2 Distributed Hash Table Based Techniques

Distributing the Namespace on multiple namenode servers has challenges. As Namenode server needs to coordinate with datanodes for block status, job execution and data blocks operations. Distributed namespace load resolves the issue of namenode single point of failure and provides scalability to namespace.

Mainly, there are two approaches to distribute namenode load:

i. Hierarchal
ii. Distributed hash table

2.2.1 Hierarchal

It is a tree based approach. At the root of the tree, there is a super namenode that coordinates among other namenode for namespace distribution and its load. This approach has changed the master control of HDFS. The system no longer relies on single namenode server. Still, this approach is semi-central. Most of the operations are coordinated by super Namenode server which is root of this distribution. The scalability of this approach depends upon the scalability of super Namenode server. To distribute namenode server load, a structured, decentralized, self organizing and self healing approach is required. One such approach is DHT.

2.2.2 Distributed Hash Table

A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes. The goal of a DHT is to allow anyone to find the node that corresponds to a given key. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

The principal idea behind DHTs is the same as for hash tables: to be able to insert, and delete large amounts of information associated with keys [13]. The concept of “bucket” is replaced by a physical node, which could be located anywhere globally. Forming an

overlay network using DHTs appears similar to the way standard hash tables are structured:

- Values are produced and associated with keys.
- Keys are hashed; in many cases with the node's port number and the node's IP address.
- The hashed keys are mapped (routed/bound) to the nodes in a balanced and procedural manner depending on the DHT algorithm (Chord, Symphony, Viceroy) that is exercised.
- The nodes hold the key and value pairs.

Collisions take place when key-value pairs are assigned to the same hash bucket. There are several collision resolution schemes that can be applied to aid sorting the keys to distinct buckets. One is known as "chaining", which employs linked lists, and the other is "open addressing", which uses probing techniques.

DHTs characteristically emphasize the following properties:

- Autonomy and Decentralization: the nodes collectively form the system without any central coordination.
- Fault tolerance: the system should be reliable even with nodes continuously joining, leaving, and failing.
- Scalability: the system should function efficiently even with thousands or millions of nodes.

Due to above characteristics of the DHT, overlay network based on DHT may be used to distribute the namespace of single namenode. There are many structured, decentralized and peer to peer approaches like Pastry from Rice, Microsoft, CAN from UC Berkeley ICSI/ICIR, Chord form UC Berkeley MIT, Tapestry from Berkeley, Symphony, Viceroy and Kademlia. Chord and Symphony DHT overlay networks are studied and distribute the work load of namenode server based on these.

2.3 Chord

The Chord protocol was designed as a DHT routing protocol by a group of students from the University of California at Berkeley and MIT [14].The designers of Chord proposed a new way to distribute data and to perform lookups, assign node IDs and keys with many attractive features such as good scalability, complete decentralization, efficient load balancing, and simplicity.

Chord maps a key to a particular node and with consistent hashing; helps distribute keys to the managing nodes in a balanced and proficient mode. Chord is an overlay protocol that has a ring structure and a deterministic topology [14].

A chord is a sequence of numbers arranged in a logical ring, 0, 1, 2 ... *n*, and looping back to 0. A (key, value) would be stored at a node that matches hash (key). If there is no node at that position, the next node ahead of that number is responsible for storing the data. This is the next node you hit if you traverse the ring clockwise starting from that hash (key) position. This node is called the successor node.

For routing queries, a node only needs to know of its successor node. Queries can be forwarded through successors until a node that holds the value is found. To increase performance and avoid the worst case of traversing the entire circle, nodes may maintain a table containing additional routing information about other nearby nodes. When a node needs to locate one that's farther away, it contacts the furthest node that it knows about whose ID is less than the hash(key) and asks it to locate the node for hash(key). The process can be repeated recursively to locate the node.

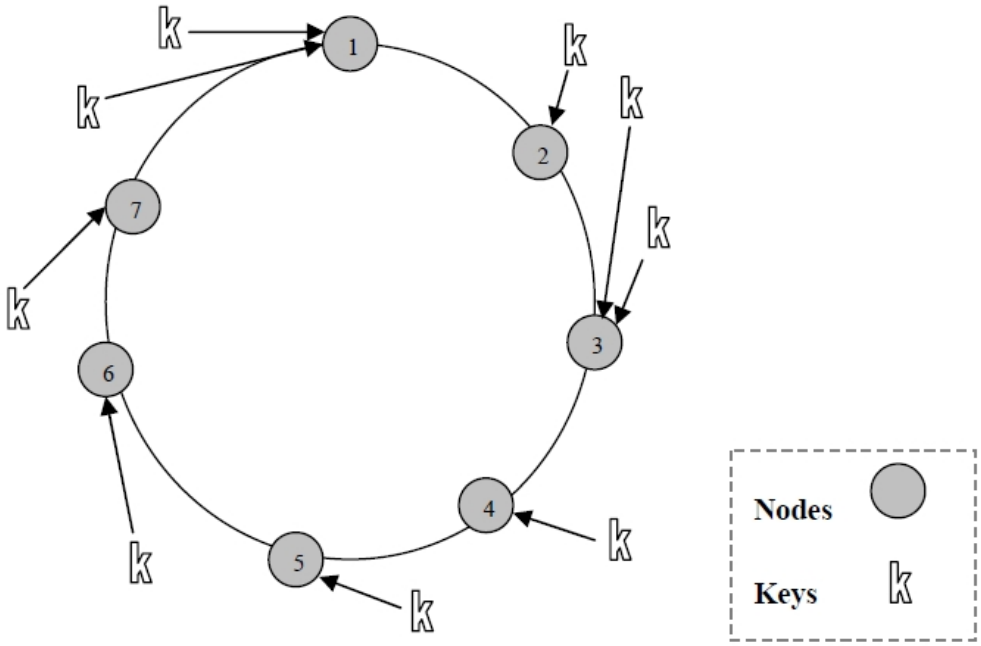

Figure 2.4 Diagram of Chord nodes and keys on the ring [14]

Load balance is another topic that is at the heart of routing protocols. Chord usually uses SHA-(Secure Hash Standard, 1993) as its principal hash function for hashing and subsequently mapping node IDs and keys to nodes thus helping load balancing.

Chord node maintains a small routing table, known as the finger table[16,17]. Finger table helps to perform lookups quickly and to handle departures or joining of nodes.

However, there is also a drawback: At times the Chord ring separates into distinct entities and does not realize that a partitioning has occurred, continuing its routing to only the nodes that it believes have survived.

2.4 Symphony

Though chord and symphony resemble each other in their common ring infrastructure, but are dissimilar in their routing techniques.

System Model

Symphony is an advanced overlay network protocol that has a ring structure using k-Long-range links to look up with its neighbor in a randomized fashion [14]. It has a strong and powerful scaling technique; it remains stable in extreme stresses.

Shown below in Figure 2.5 is a model of a Symphony network that displays its short-distance links, the reinforced short-distance links, as well as the long-distance links.

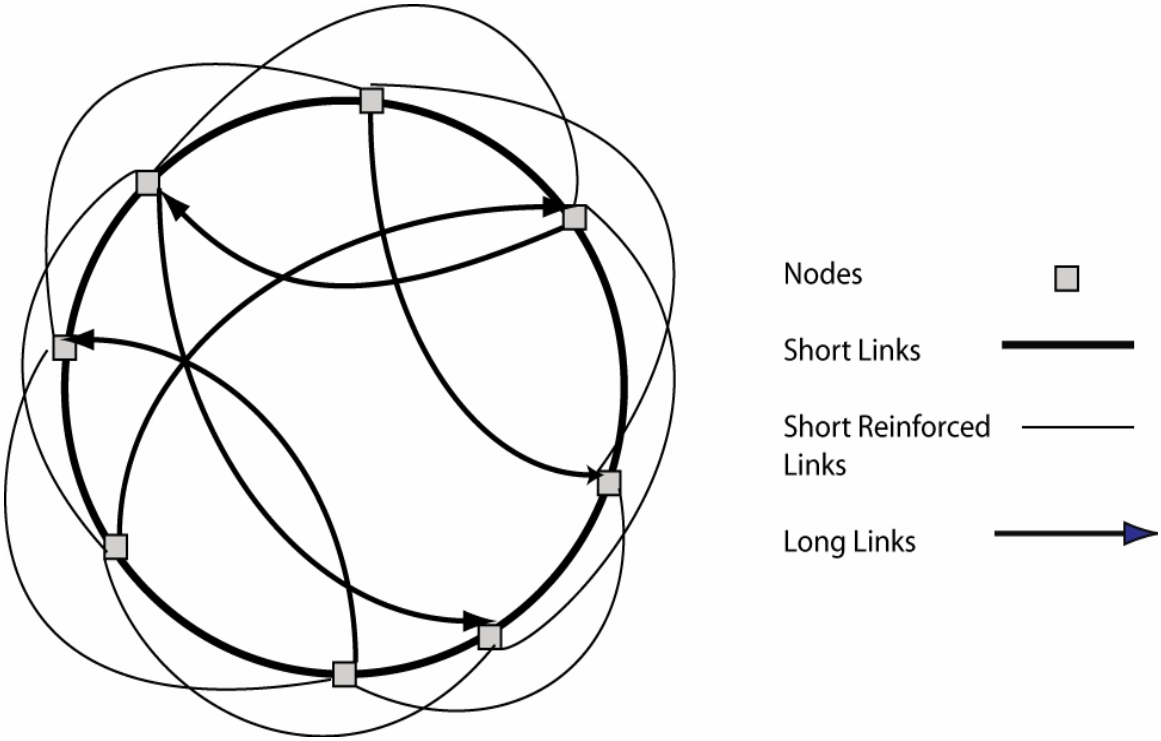

Figure 2.5 Simple example of a Symphony network with short reinforced links and one long distance link per node (original design by Manku *et al.*, 2003)[14]

Symphony logical overlay handles a large load of dynamic hosts and provides the basic services of inserting, deleting, and looking up the hashed keys that store information at node level.

Symphony breaks large routing tables into more manageable blocks that can then be managed by local nodes. Information can be quickly accessed, deleted, and inserted.

Symphony has very few keep-alive messages for nodes that are departing or inaccessible. This minimizes the traffic usually generated by message exchanges between the routing tables and creates room for faster operations.

The Symphony model only backs up the short-range links from the predecessors and to the successors in the ring topology but excludes any backups for the long-distance links. Symphony uses the long-distance links to create short paths to the desired target. This also diminishes the amount of traffic in the overlay and generates more space for the free flow of information.

In Chord, finger tables are not flexible, but grow logarithmically with the number of nodes joining or leaving the system. But with Symphony the number of links is not an issue. Symphony's structure is so versatile that it can adjust to any number of links.

Symphony uses a special feature that the protocol starts without specifying the maximum node numbers and can be adjusted afterwards. This is the only P2P protocol that offers this feature. Symphony's efficiency is due to another feature known as 1-lookahead.

2.4.1. Greedy Routing

Greedy Routing is to make choice based on immediate rewards rather than looking ahead to see the optimum. The best way to get to a destination is without other knowledge, going down the road that heads in the direction of the destination is probably a reasonable choice. This is the greedy approach and can do well in certain cases. It also makes the decision much easier than considering all alternatives. It may not be the optimal decision, in contrast to considering all possible alternatives and then subsequent alternatives, etc

So, Greedy routing is a novel routing paradigm where messages are always forwarded to the neighbor that is closest to the destination. It follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

2.4.2 Advantages of Greedy Routing:

- Approximates optimal solution
- May or may not find optimal solution

- Provides “quick and dirty” estimates
- A greedy algorithm makes a series of “short-sighted” decisions and does not “look ahead”
- Spends less time

2.4.3 Greedy Routing with 1- lookahead Feature in Symphony

Symphony [15] is a successful adaptation of Kleinberg's construction [15] to arrive at a randomized P2P routing network. The idea is to place nodes in a ring and to equip each node with multiple long-distance links. An adaptation of Kleinberg's construction is appropriate for P2P routing because individual nodes have low degree and yet are able to route messages efficiently using only local information.

An important distinction between deterministic and randomized networks pertains to information available to different nodes about the global network. In deterministic networks, each node has knowledge of the entire network. The structure of the network is global information. Therefore, messages can be sent along shortest paths. However, in the randomized networks, each node possesses knowledge of only its own set of long-distance links. Each node makes links as a function of some random bits. It is assumed that these random bits are not global information. Therefore, it is not possible to send messages along shortest paths, in general. This motivates the need for decentralized routing strategies which allow a node to forward messages on the basis of as little knowledge of other nodes' random bits as possible. Greedy routing is a natural decentralized routing strategy: a node forwards a message along that out-going edge that minimizes the distance remaining to the destination.

Greedy Routing with one lookahead feature is used in Symphony namenode servers i.e. maintaining short distance links with its immediate neighbors along the circle and also establishing 1 long distance links.

2.5 Comparison of Chord and Symphony

- N is the maximum number of nodes in a ring. In chord, each node maintains a routing table based on N. This table is called Finger table. So, Chord depends upon N which

might not determine at start. Symphony does not specify this initial parameter and adjust afterwards.

- Sometime, Chord makes partition among nodes and continues working in same way because it learnt from predefined values of maximum nodes.
- Chord generates internal load of maintaining finger table for each node. It has compulsion to maintain the state of each machine in finger table. It periodically sends the status request to these machines. This generates traffic and in large network it becomes a overhead on network performance.
- Symphony-style greedy routing over bidirectional links that minimizes absolute distance to the target at each hop. Chord uses a rather expensive re-linking and stabilization protocol upon every insertion and deletion in finger table.

Symphony is preferred over Chord because [15]

I. It has low state maintenance which means:
   - Low degree : Fewer pings/keep-alives, less control traffic
   - Low degree :Distributed locking and coordination overhead over smaller sets of nodes
   - Low degree : Smaller bootstrapping time when a node joins ,Smaller recovery time when a node leaves

II. Fault tolerance :
   - Only short links are bolstered. No backups for long links

III. Smooth out-degree vs. latency tradeoff :
   - Only protocol that offers this tuning knob even at run time!
   - Out-degree is not fixed at runtime, or as a function of network size.

IV. Flexibility and support for heterogeneity
   - Different nodes can have different #links

2.6 Key Distribution Techniques

Mainly there are three key distribution techniques:

2.6.1 Fixed Key Distribution: In Fixed distribution technique, the namespace object ids are mapped to namenode in a predefined range. Any key falling under a specified range is to be managed by that namenode server irrespective of its location and load.

A DHT contains key and its value. This distribution assigns keys of specified range to different nodes and nodes stores the values for all the keys for which it is responsible. It adopts key lookup table to partition the files among different nodes. It is possible that multiple object ids, which are dynamically generated, may belong to a particular range so the namenode which is catering the namespace service is imbalancing the namespace load. This scenario is even worse when billions of keys are generated. It generates load on that particular node while other nodes sit idle leading to load imbalance among all the nodes. The idle key distribution should distribute keys evenly among all the nodes.

This technique is simple and is good in a scenario where keys are small in number, leading to less uneven distribution of keys. Another important issue is that when a new node joins, a new range is assigned to that namenode and key migration and shuffling happens. Reshuffling keys again is a time consuming and costly process.

2.6.2 Uniform Key Distribution: In uniform distribution, modulus arithmetic operation is used. If there are N number of nodes and K are the number of keys then this key resides on namenode K modulus N. This technique distributes keys evenly but it relies on fixed initial keys.

Keys are assigned to different namenodes according to modulo operation. In dynamic key allocation, the key ids are generated sequentially and are equally divided among namenodes. So, it can evenly distribute even billons of keys and there is no load imbalance on particular node.

This technique is simple and is good in a scenario where keys are of any number. The issue of addition of new namenode remains as in fixed distribution. The key migration and shuffling also happens. Reshuffling keys again is time consuming and costly. The

uniform distribution only solves the problem of skewed load distribution of keys [18]. Other challenges like churning of namenode exert pressure on the resources.

2.6.3 Random Key Distribution: In this technique, the keys are to be distributed randomly [19] to namenode servers. This technique is quite simple but has uneven distribution of keys leading to uneven load distribution. The issue of namenode churning requires key migration and is costly.

There is a need of a namespace distribution technique that ensures even distribution of keys, minimum migration and usage of resources while considering namenode churning and locality of keys. The locality of keys means that the keys of a particular geographic region are available locally to that region and its replication are kept at different region to provide high availability.

There is a definite need of an overlaying network structure for namenodes placed geographically which has above mentioned characteristics and is highly scalable. The ideal technique would be to consider locality of data *i.e.* data is available within local domain. So, geographic based distribution may be developed for the distribution of keys.

2.6.4 Geographic Based key distribution: Geographic hash table hashes keys into geographic coordinates, and stores a key-value pair at the name node server geographically nearest the hash of its key [20, 21]. It uses an efficient consistency protocol to ensure that key-value pairs are stored at the appropriate nodes and it distributes load throughout the network using a geographic hierarchy. A data object is associated with a key and each node in the system is responsible for storing a certain range of keys. By hashing keys, GHT spreads storage and communication load between different keys evenly throughout the namespace.

Chapter 3

Problem Statement

3.1 Problem Definition

HDFS is based on single-node namespace server architecture. Since the name-node is a single server of the file system metadata, it is one of the limitations for file system growth. In order to make metadata operations fast, the name-node loads the whole namespace into its memory, and therefore the size of the namespace is limited by the amount of RAM available to the name-node.

The namespace consists of files and directories. Files are divided into large (128 MB) blocks. To provide data reliability, HDFS uses block replication. Each block by default is replicated to three data-nodes. Once the block is created its replication is maintained by the system automatically. The block copies are called *replicas*. The name-node keeps the entire namespace in RAM. This architecture has a natural limiting factor: the memory size; that is, the number of namespace objects (files and blocks) the single namespace server can handle.

As the data is increasing day by day *i.e.* in petabytes or zettabytes of data so namespace is also growing. 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 [10] 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. 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 cannot store data more than 60 petabytes.

The namenode server has issues of namespace scalability, failover and DOS (Denial of service). In case, an application is using heterogeneous data lying in cities, countries and continents then it becomes huge challenge to manage such a single big namespace of objects and it is not possible for a single or few namenode servers to manage such a huge namespace in memory.

So it needs to relook and redesign the approach to manage the namespace for such applications that addresses the scalability, provide high availability and reliability, and ease of access and at the same time maintaining the locality of reference.

Objectives are:

i. Comparison of structured DHT Symphony and Chord on the basis of network creation time and key lookup time.
ii. Design a new approach for geographical based namespace distribution using Symphony to create a highly scalable environment for namespace and its performance comparison with different number of namenodes.
iii. Designing a Key Lookup Algorithm in Symphony using Neighbor of Neighbor Greedy Routing Approach with 1 Look Ahead.
iv. Comparison of key lookup time on multiple namenode servers with different number of nodes with a single namenode server.

3.2 Methodology

The step-by-step methodology is followed in creation of symphony overlay network for multi namenodes servers, randomly distributing the keys of namespace on these nodes, and analyzing scalability, availability and lookup time is as given below:

i. Comparative analysis is done between DHT Chord and Symphony.
ii. Namespace has been distributed on multiple namenode servers using Symphony
iii. Planetim Simulator is used for simulation of Symphony. B+ trees are implemented to compare the lookup time of single namenode with multiple name nodes.
iv. Algorithm is designed for key lookup from distributed namespace on multiple namenodes using symphony.

Chapter 4

Distributed Hash Table Based Technique

As the data is growing day by day, one of the important issues which need a consideration is that the namespace is growing and it requires huge amount of primary memory. The growing size of namespace imposes big challenge to achieve high availability and scalability as there are millions of clients who want to access the files and the files too are billion files, so the issue is scalability of namespace and the namespace is only master namespace so other issue is single point of failure. If the single namespace fails then the issues arises is how to recover the whole namespace which contains billions of files and directories. The mean time to recover for Namespace system is very large with a single namenode server. So, Namespace distribution techniques are required to address the issue of scalability and single point of failure. Some approaches are presented which addresses the issues of single namespace server. In such a system structured, decentralized, self-healing, self-organizing approach is required to address this issue.

The proposed approach which is based on the distributed hash table resolves the issue of single point of failure and improves scalability, reliability, availability and load balancing.

4.1 Distributed hash table based technique

Distributed Hash Tables (DHT) are used to distribute the namespace. DHTs [13] is a class of a decentralized distributed system that provides a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes.

The goal of a DHT is to allow anyone to find the node that corresponds to a given key by using a flexible architecture to store large amount of data on namespace servers and balance the load on the nodes. There are few key distribution techniques which help to distribute a big namespace to multiple namenode servers.

4.2 Namespace based distribution using Distributed Hash Tables

Distributed Hash Tables divides the namespace into multiple namenode servers in decentralized manner. So it is not prone to single point of failure. Namespace is distributed on different namenode servers and are replicated on different namenode servers for load balancing and to achieve high scalability. The main aim of building such a system is to cater the growing demand of namespace and to support high availability and scalability. Distributed hash table divides the namespace of HDFS to multiple namenode servers. The current namespace limit is 100 million files. Static partitioning allows it to scale the federated namespace to billions of files. Estimates show that implementation of a dynamically partitioned namespace supports 100 billion objects. DHT is a managed and structured approach for the scalability of HDFS. Availability is another strong motivation for the distributed hash table based namespace.

There are structured, decentralized and peer to peer approaches like Pastry from Rice, Microsoft, CAN from UC Berkeley ICSI/ICIR, Chord form UC Berkeley MIT, Tapestry from Berkeley, Symphony, Viceroy and Kademlia.

Considering all above mentioned concerns, the thesis proposes an improved Hadoop system approach that provides dynamic distribution of namespace to achieve high scalability, availability and guard the system from single point of failure. The design not only eliminates the limitations but also improve the Hadoop core functionalities.

4.3 Proposed Approach of Namespace Using Symphony

The namenode servers form a ring and namespace is distributed on the namenode servers. This is approach is an improvement over the default Hadoop which is prone to single point of failure.

The proposed approach of Namespace using Symphony is described figure 4.1:

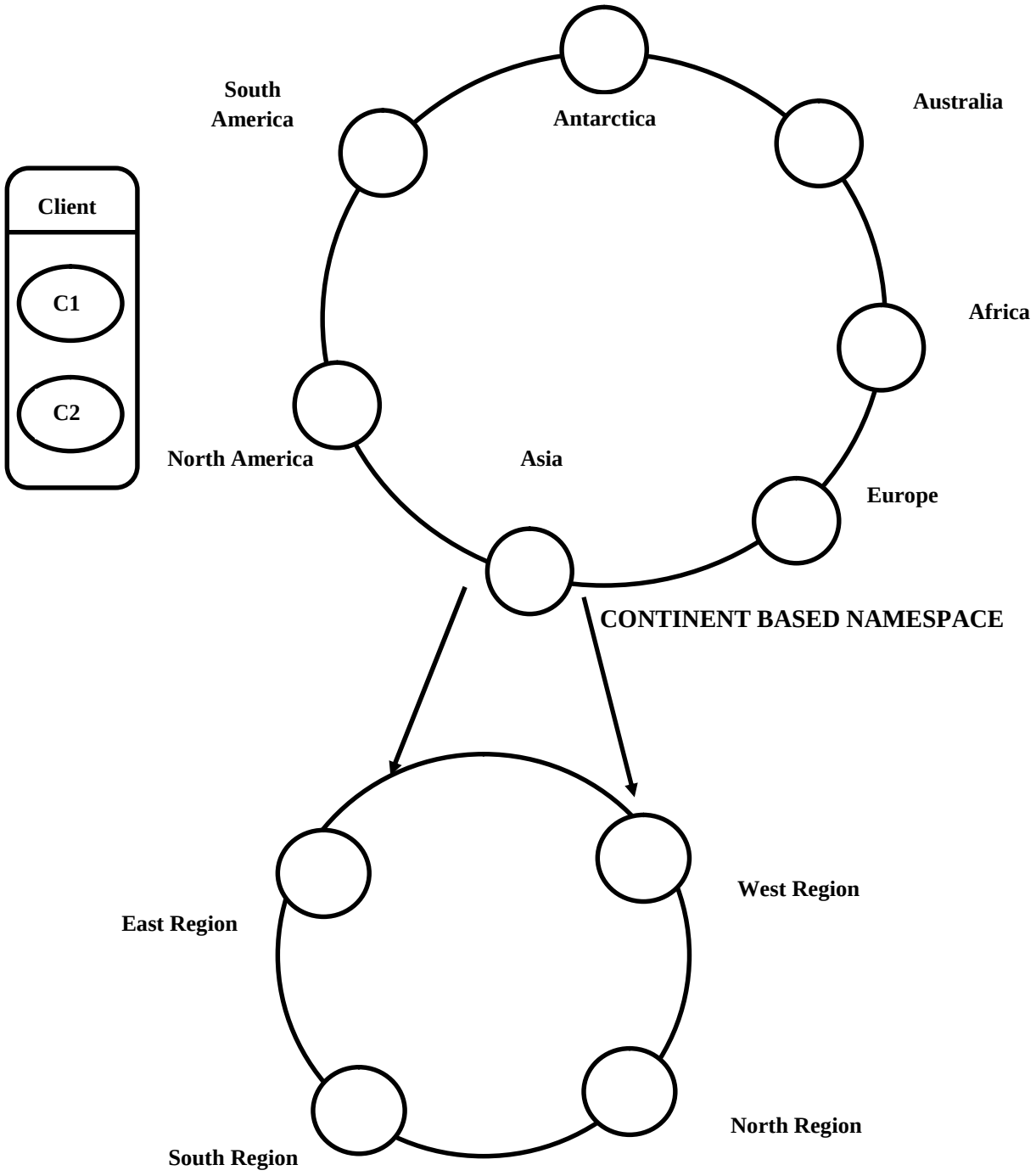

Figure 4.1: Geographic Based Namespace Distribution

The proposed approach is geographically based namespace distribution where keys are distributed on namenode server. The namenodes are connected through symphony

overlay P2P network as shown in figure 4.1. Symphony is used as it is efficient in utilizing the WAN links for key lookups. It decreases the usage of network bandwidth and maintains the data locality for namespace. The number of nodes may be divided on region based i.e. east, west, north and south and the connection between number of nodes is on WAN/LAN links. Symphony optimizes bidirectional routing *i.e.* both incoming and outgoing links, route to neighbor that minimizes absolute distance to destination and also it reduces average latency as it maintains list of neighbors of neighbors.

Chapter 5
Implementation and Results

5.1 PlanetSim Simulator

The primary component of the design approach is distributed name node servers in symphony ring. To evaluate the performance of proposed design approach, PlanetSim simulator is used to construct the Symphony ring for name servers and to compare the Chord and Symphony based distributed hash tables Protocol Implementation of simulator is done on Ubuntu 12.04 LTS 64 bit with kernel 3.2.0-41 with 2 GB of RAM and Intel Core i32329M CPU @2.20GHx4 Processor.

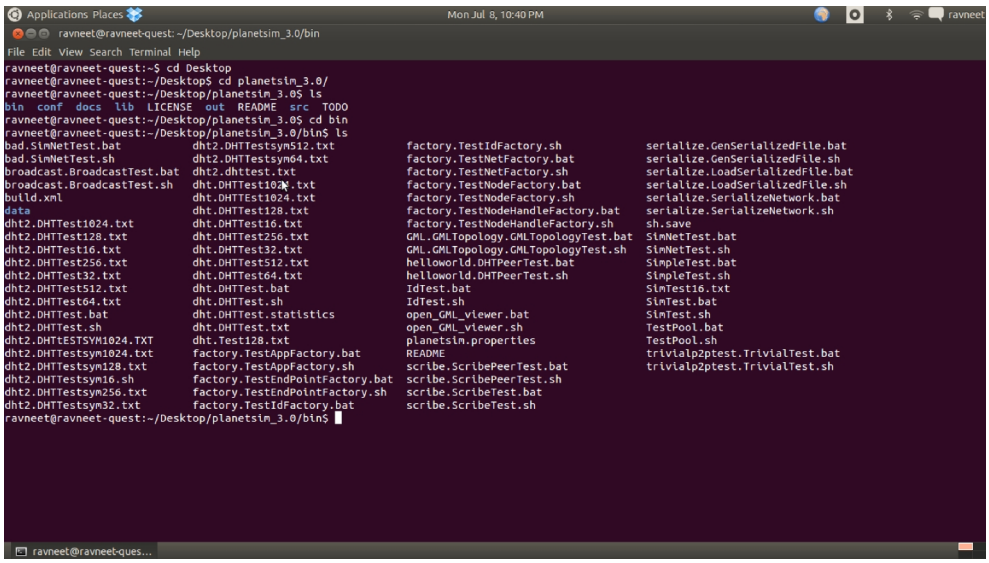

Figure 5.1: PlanetSim Simulator

PlanetSim is an object oriented simulation framework for overlay networks and services. This framework presents a layered and modular architecture with well defined hotspots documented using classical design patterns. PlanetSim has been developed in the Java language. PlanetSim is an innovative Java based P2P simulator and is used to tests DHT based overlays like Chord and Symphony. Its mainframe is modular in structure and can easily be extended to include other tests without penalizing the entire infrastructure. This

is the reason why, in this thesis project, PlanetSim was chosen to conduct all the simulations for Symphony and Chord as well as for testing new and novel ideas.

To validate, Symphony based approach is implemented for Namespace load distribution and balancing in Hadoop. This simulator helps in designing improved Symphony protocol that considers the geographical data locality features for Namespace load distribution and balancing in Hadoop.

PlanetSim Simulator is used to do the comparison of Chord and Symphony ring. The parameters for chord and symphony ring are defined in PlanetSim. The conf files are modified according to the requirement of the design approach. The parameters like number of nodes, topology, DHT overlay type and 1000 sample keys are used. The result of tests are obtained such as Network creation time, number of steps, key lookup time as shown in table 5.1 to 5.2.

The following results are found in DHT2 Chord and Symphony ring.

Table 5.1: Chord Network creation time and key lookups against different number of nodes

DHT2 Chord				
	Network Creation		**Four key lookups**	
Number of Nodes	**Time (s)**	**Steps**	**Time (s)**	**Steps**
16	0.151	937	0.048	1210
32	0.219	1115	0.039	1398
64	0.507	1435	0.026	1737
128	0.613	2075	0.027	2385
256	1.181	3355	0.040	3668
512	2.754	5906	0.064	6224
1024	9.870	11035	0.139	11354

The table 5.1 shows the network creation time, four key lookups for different number of nodes ranging from 16 to 1024 in DHT2 Chord.

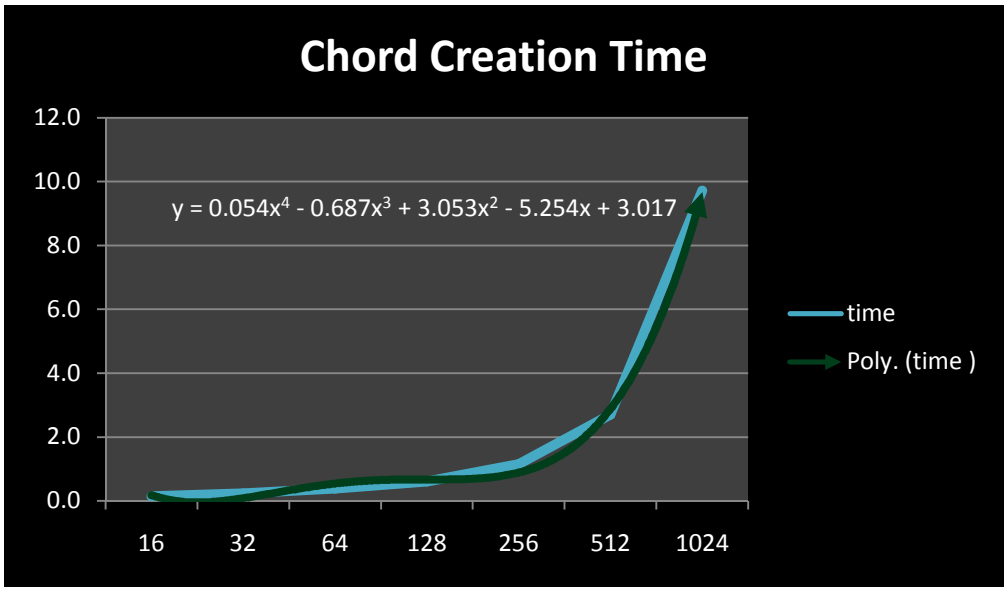

Figure 5.2: Chord Creation Time (in second) vs. Number of nodes

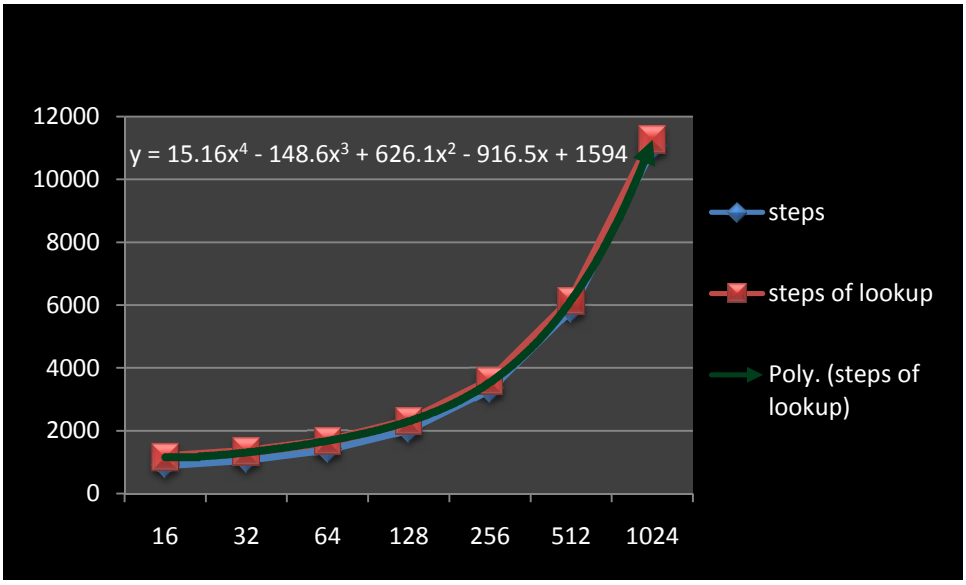

Figure 5.3: Number of steps/ Lookup steps vs. number of nodes in Chord

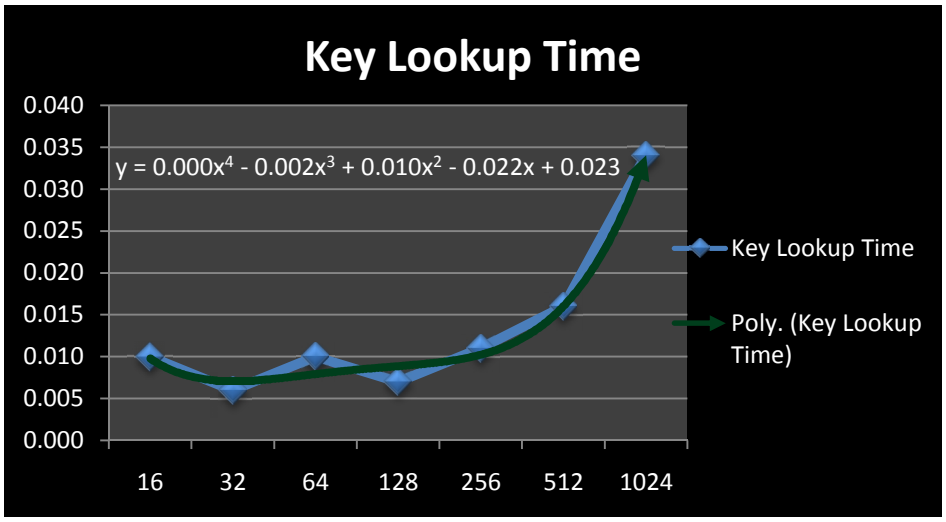

Figure 5.4: Chord Key lookup time (in seconds) vs. number of nodes

The figure 5.2 to figure 5.4 shows the Chord creation time, number of steps taken and the key lookup time taken against number of nodes. The growth function of all these curves is presented by a polynomial of degree 4.

Table 5.2: Symphony Network creation time and key lookups time against different number of nodes

DHT2 Symphony				
	Network Creation		Four key lookups	
Number of Nodes	Time (s)	Steps	Time(s)	Steps
16	0.040	89	0.011	313
32	0.086	136	0.016	362
64	0.194	213	0.015	443
128	0.347	388	0.017	621
256	0.557	742	0.014	980
512	1.037	1414	0.022	1660
1024	3.013	2857	0.044	3111

The table 5.2 shows the network creation time, four key lookups for different number of nodes ranging from 16 to 1024 in DHT2 Symphony ring.

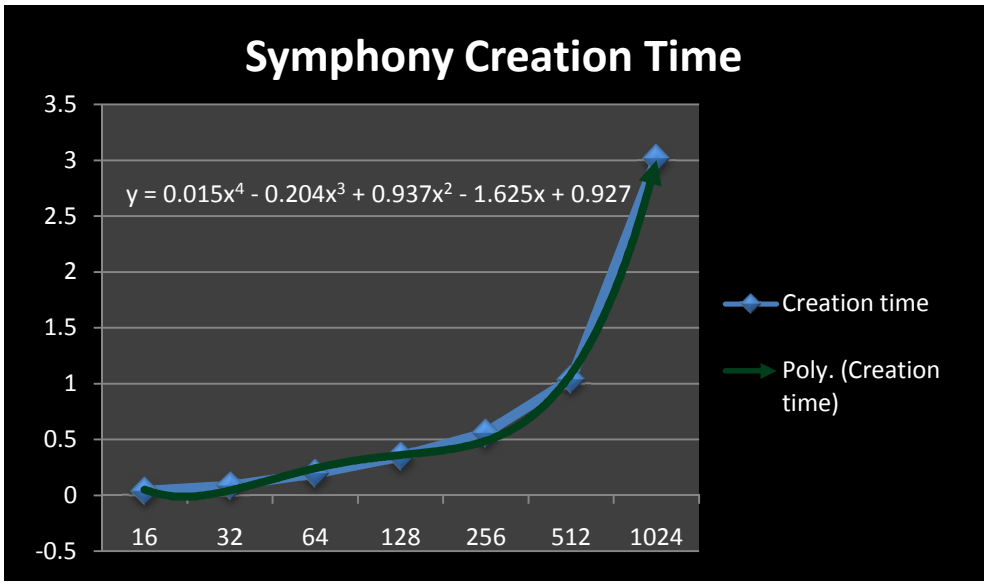

Figure 5.5: Symphony creation time vs. number of nodes

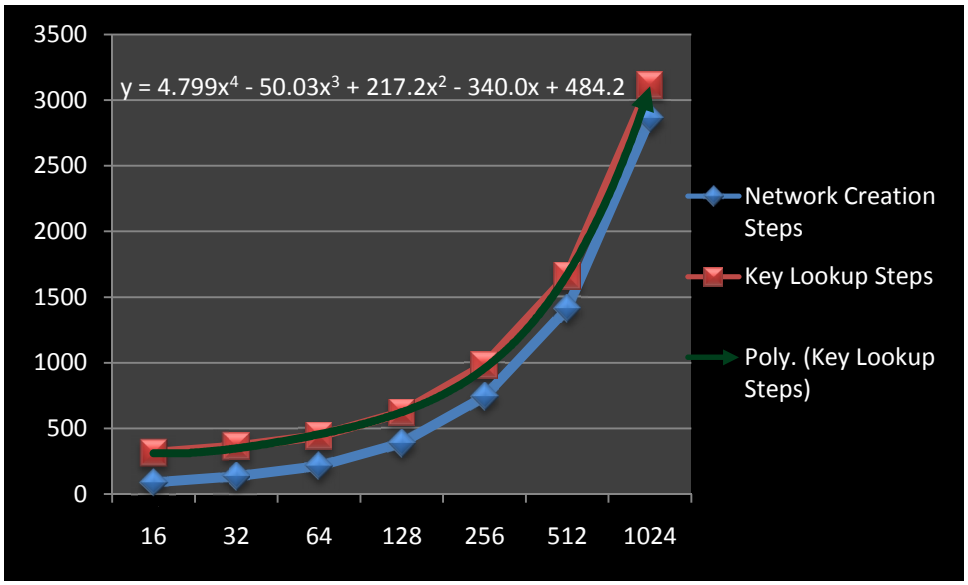

Figure 5.6: Number of steps/Lookup steps against different number of nodes in Symphony

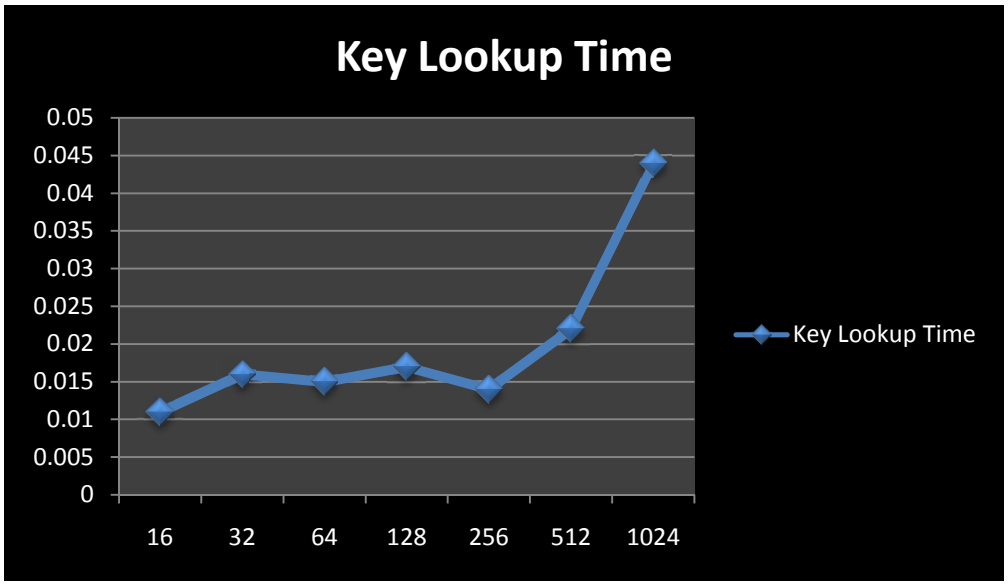

Figure 5.7: Symphony Key Lookup time (in seconds) against different number of nodes

The figure 5.2 to figure 5.4 shows the Symphony creation time, number of steps taken and the key lookup time taken against number of nodes. The growth function of all these curves is presented by a polynomial of degree 4.

From the results shown in all the figures from 5.2 to 5.7, it is infer that the network creation time in symphony is far less than that of chord ring because Symphony is bidirectional in nature i.e. node has incoming and outgoing links [22] both sides and it has 1 look ahead feature *i.e.* node could inform each other about position of their respective long distance links in a ring. Thus node can learn and maintain a list of its neighbors of neighbors. It helps in getting closer to final destination.

The number of steps taken in Symphony is also quite less than that of chord ring. Less number of steps means less traffic and less bandwidth utilized. Low degree network reduces the number of network connections and ambient traffic corresponding to ping, keep alive and control information. But the time required in creating a chord ring of name node servers and their lookup data structures depends upon finger table, successor list and predecessor list.

The growth function of y=f(x) where x is number nodes in symphony and is shown in the figures5.2 to 5.7. Key insertions and key lookups are quite fast, with simulation time about linear with number of nodes. In all cases, the Symphony simulations run significantly faster than with Chord. Key insertions and lookups with Symphony are again simulated quicker than for the Chord

5.2 Key Lookup Algorithm in Symphony using Neighbor of Neighbor Greedy Routing Approach with 1 Look Ahead

Let us assume that client request for a key lookup to node S (source) such that {S | S≠D} where NL with number [(s+1) mod n] be the left, NR with number [(s-1) mod n] be right neighbors of node S respectively and D is destination.

In case of key look up failure from S, NL and NR, request for key is routed to 1 long distance call which is a random number r from the probability distribution p(s) = s / log n where s ∈ [1, n] and then establishes a link with node [s + r] mod n for source node s. In the mean time, NL, NR will repeat the search and pass the key lookup request to right neighbor of NR and Left Neighbor of NL. This will continue until key is found and sent back to the requesting node S.

Local_Lookup (node_number, key, s)
// node_number represents node number where key will be searched locally using //B+ three search, key represents a key to be searched for its corresponding value //and s is the source node number.

```
{
Search in local keys stored in B+ trees of Node NR
        if (key found in s)
        return (value,s) // return value to source node s
        exit
}
```

```
Symphony_key_lookup (s,n,key)
// s represents source node number, n is number of nodes and key represents a key
//to be searched for its corresponding value
{
        Repeat while (key found)
                Local_lookup(s,key,s)
        if ( requested from s)
                nr= [(s-1) mod n];  // Right Node number of initial Node s
                Local_lookup(nr,key,s) //Local Key lookup in Node NR

                nl =[(s+1) mod n];  // Left Node number of initial Node s
                Local_lookup(nl,key,s) //Local Key lookup in Node NL

                r =s / log(n);
                long_distance_node_number=[s+ r] mod n
                Symphony_key_lookup (long_distance_node_number,n,key);
                // All above call under if will run in parallel
        else
                If (nl OR nr)
                        nl =[(nl+1) mod n];
                        Symphony_key_lookup (nl,n,key);
                        r =s / log(n);
                        long_distance_node_number=[ ceil (nl+ r)] mod n
                        Symphony_key_lookup (long_distance_node_number,n,key);
                        nr =[(nr-1) mod n];
                        Symphony_key_lookup (nr,n,key);
                        r =s / log(n);
                        long_distance_node_number= [ceil (nr+ r)] mod n
                        Symphony_key_lookup (long_distance_node_number,n,key);
                        // All above call under if will run in parallel
}
```

The key lookup time can further be increased by increasing the number of look ahead with additional cost resources.

5.3 Comparison of key lookup time on randomly distributed keys on multiple namenode servers with key Lookup of single namenode server

To compare the key lookup time on multi namenode servers with single namenode server, B+ trees are implemented in python and keys are generated randomly for single namenode servers and multi namenode servers. The program is executed on windows 7 with i5 processor 2.53 GHz and 4 GB RAM. The experimental results are collected for hundred key lookup, average lookup time and mean time to recover (MTTR) of the system. The results are shown in table number 5.3 to 5.6

Table 5.3: Mean Time to Recover in Symphony

Number of Keys (in millions)	Number of nodes	Mean Time To Recover (MTTR) in sec
50	1	24.523

The single namenode server managing 50 millions of keys take 24.523 seconds on an average to recover the system from any failure and is also called mean time to recover (MTTR). The mean time to recover is a time to fetch all metadata from hard disks to memory or cache of namenode system at startup. Higher the MTTR affects the availability of system. System remains unavailable during that time for client queries. The meaning of MTTR is same as startup time.

Table 5.4: Hundred key lookup time and Average time in Symphony

Number of Keys (in millions)	Number of namenode server	Hundred Keys Lookup Time in sec	Average Time Per Lookup in sec
50	1	72.3080	0.7231

The 50 million keys are generated randomly for single namenode server and out of which hundred keys are taken randomly and are searched. The total time to search hundred keys is approximately 1.867 seconds with average lookup time 0.4665 seconds.

Table 5.5: Mean Time to Recover on Multi Namenode Servers in Symphony

Number of Keys (in millions)	Number of nodes in Symphony	Mean Time To Recover (MTTR) in sec
50	16	1.8500
100	16	3.7390
150	16	5.0700
200	16	7.7200
250	16	9.7300
300	16	10.7100
50	32	0.9410
100	32	1.9800
150	32	2.4800
200	32	3.7200
250	32	4.0100
300	32	5.9100
50	64	0.3900
100	64	1.0820
150	64	1.4080
200	64	1.9800
250	64	2.3400
300	64	2.7700
50	128	0.2500
100	128	0.4400
150	128	0.7300
200	128	1.0000
250	128	1.1900
300	128	1.3040

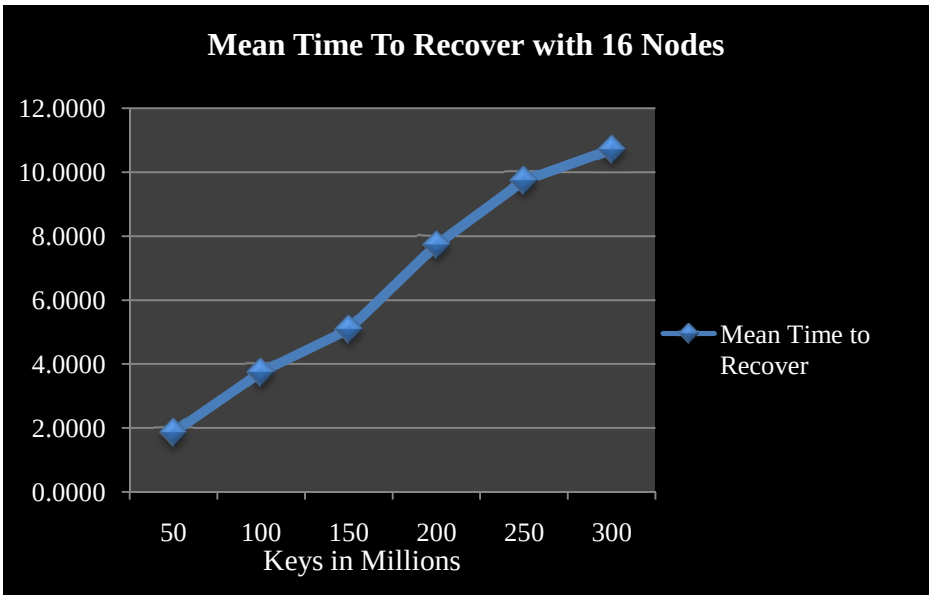

Figure 5.8: Mean Time to Recover With 16 Nodes

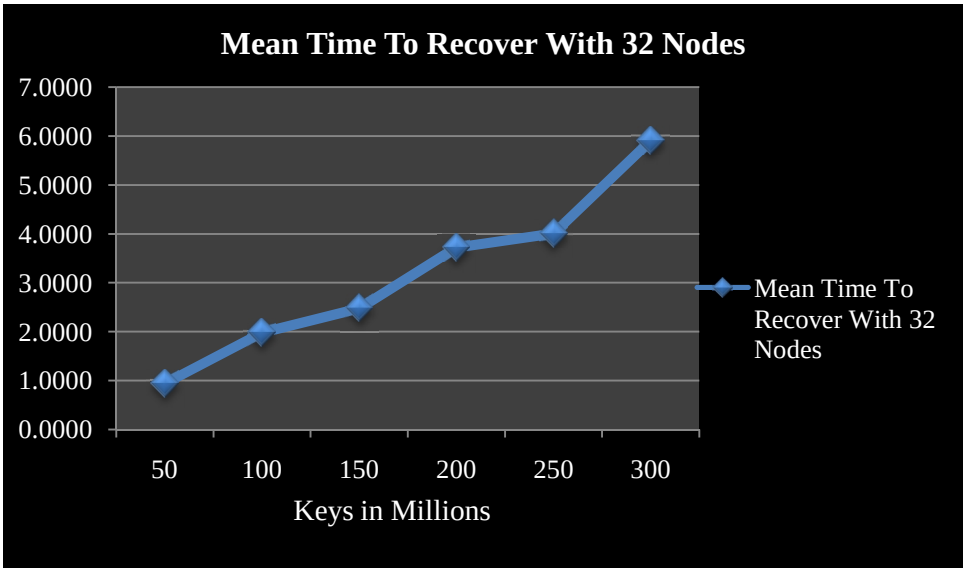

Figure 5.9: Mean Time to Recover With 32 Nodes

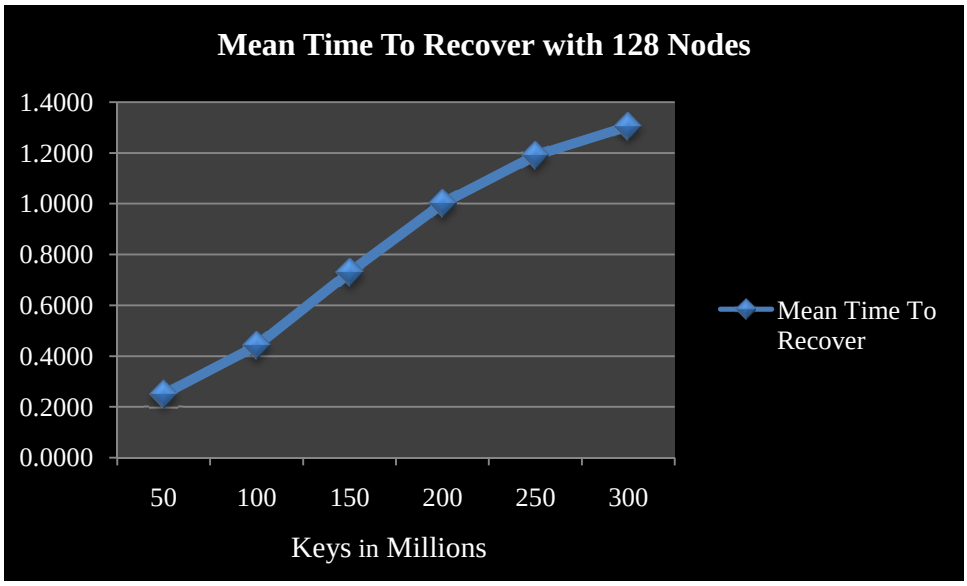

Figure 5.10: Mean Time to Recover With 128 Nodes

The figure 5.8 to figure 5.10 shows the Mean time to recover with different number of nodes in Symphony.

The multi namenode server managing 50 million or more keys are distributed on multi namenode servers. The results are collected and are presented in mentioned table 5.3. There is substantial reduction in MTTR and it helps in improving the availability of the system. So our experiment demonstrates that multi namenode servers exhibit higher availability than single namenode server. The single namenode server is not able to maintain 50 million keys due to memory restriction. The results of 300 millions of keys are shown in table 5.3 and it can scale up to higher number by adding more namenode server in Symphony. So this approach has broken the limit of keys in namespace server.

Table 5.4: Symphony hundred key lookup time and Average time on Multi Namenode Servers

Number of Keys (In Millions)	Number of nodes in Symphony	Hundred Keys Lookup time (in sec)	Average Time Per Lookup (in sec)
50	16	5.3200	0.0532
100	16	11.3700	0.1137
150	16	11.0900	0.1109

200	16	15.6330	0.1563
250	16	30.7170	0.3072
300	16	33.9700	0.3397
50	32	2.8000	0.0280
100	32	5.4800	0.0548
150	32	7.2000	0.0720
200	32	8.5800	0.0858
250	32	14.7880	0.1479
300	32	16.9000	0.1690
50	64	1.4100	0.0141
100	64	3.2320	0.0323
150	64	4.9440	0.0494
200	64	5.3100	0.0531
250	64	6.1500	0.0615
300	64	9.3310	0.0933
50	128	0.8700	0.0087
100	128	1.5300	0.0153
150	128	2.3100	0.0231
200	128	2.8600	0.0286
250	128	3.8000	0.0380
300	128	4.2000	0.0420

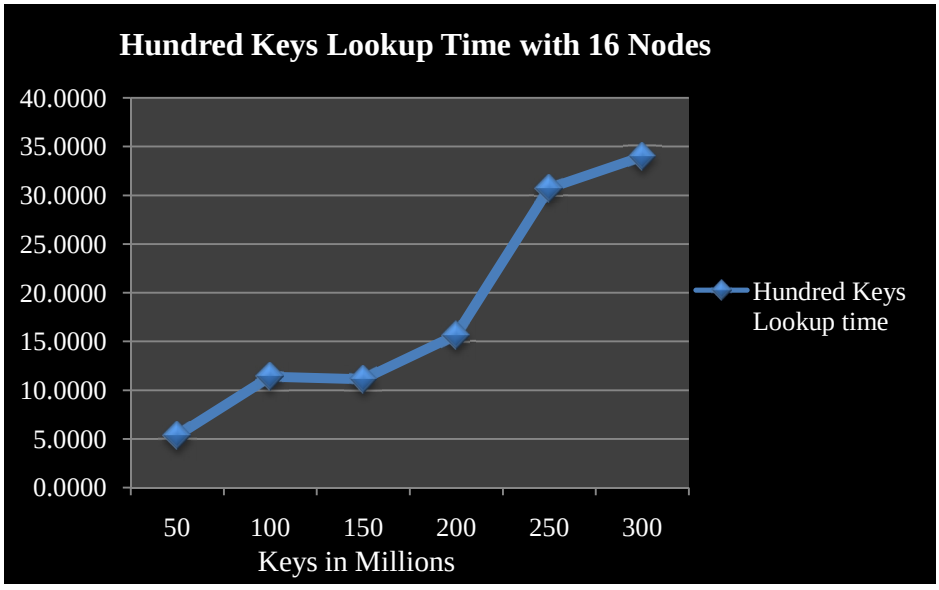

Figure 5.11: Hundred Key Lookup Time with 16 Nodes

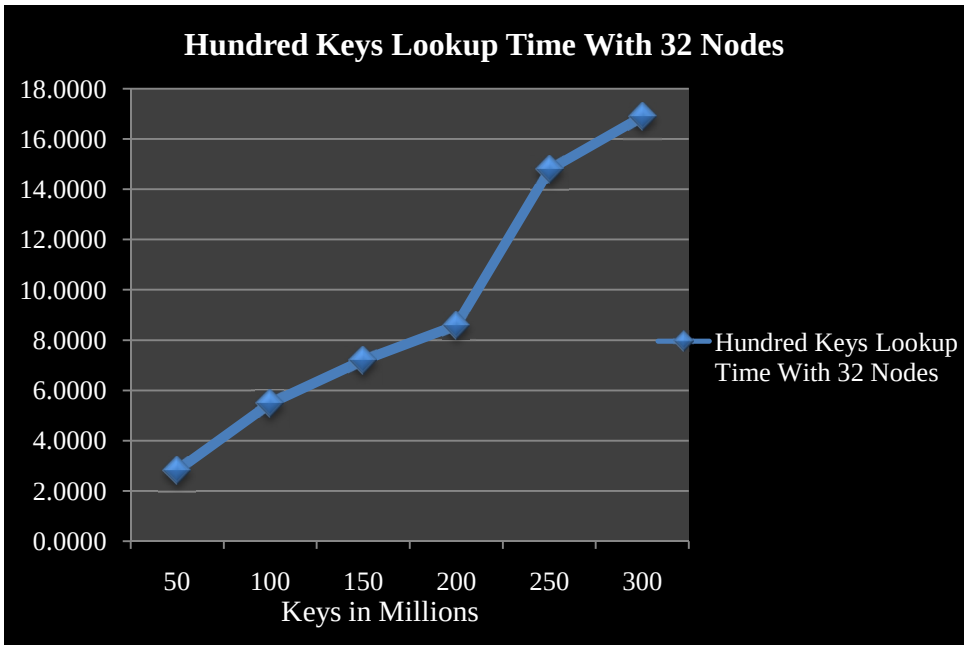

Figure 5.12: Hundred Key Lookup Time with 32 Nodes

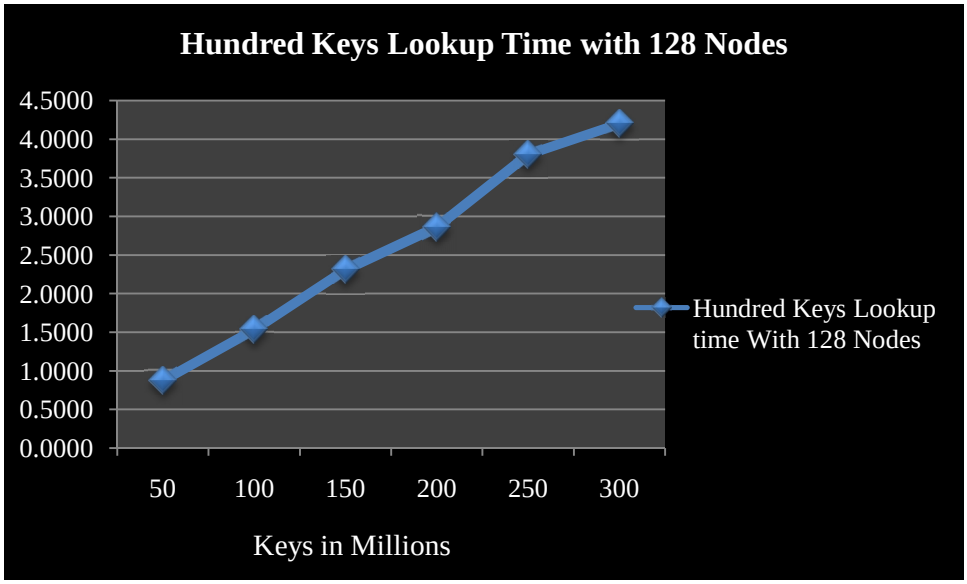

Figure 5.13: Hundred Key Lookup Time with 128 Nodes

The figure 5.11 to figure 5.13 shows the hundred Key Lookup time in Symphony with different number of nodes.

The 50 million or more keys are generated randomly for multi namenode servers and are distributed equally among namenode server in Symphony overlay network. Hundred keys are taken randomly and are searched. The total time to search hundred keys and average time to search a key is reduced substantially on multi namenode servers. It demonstrates that the response time of system with multi namenode servers improves considerably than single namenode server.

The proposed approach has shown significant improvement in performance, scalability of the system, less MTTR (high availability) and addresses the issue of single point of failure of namenode server. The proposed approach exhibits self-healing, self-organizing, self-repairing, scalable, reliable, and decentralized as it uses Symphony (DHT).

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

Hadoop Distributed File System is being widely used by many organizations for its low cost of implementation and to analyze big data sets. But limitation of Hadoop architecture *i.e*. its centralized namespace storage limits its scalability, availability and issue of single point of failure. DHT P2P structured approaches are used to distribute and balance the load of namenode servers because these approaches are self-healing, self-organizing, self-repairing, scalable, reliable, and decentralized.

The proposed approach is based on Symphony and has resolved the issues of namespace scalability, availability and address the issue of single point of failure. The focus of the work is based on geographic based namespace distribution using Symphony. The system has achieved high scalability as namespace is distributed among namenodes by using distributed hash tables. In previous approaches, the growing namespace of HDFS affects the availability and performance of Hadoop cluster.

In this research work, the performance and availability of namespace is scaled up by adding namenode server. For this purpose, Chord and Symphony are studied and their features are compared. In this comparison, it is found that Symphony is better to handle large number of nodes as it generates less traffic while looking up a key. Also it is quite flexible as it does not depend upon the maximum number of nodes in a network. Overall, Symphony scale wells with low lookup latency, low maintenance cost with few neighbors per node.

The proposed approach has shown significant improvement in performance, scalability of the system, less mean time to recover (high availability) and addresses the issue of single point of failure of namenode server.

6.2 Future Work

The performance of the system can further be improved by using adaptive strategies for replication of distributed namespace keys. The relationship between client applications and these namenode can be developed to further improve the performance of the namespace.

The client application data resides on a distributed namespace. So, it requires careful review of the security and delivery of quality of service. The resulting distributed namespace architecture may change the development directions for future large scale data storage system.

REFERENCES

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

[2] Apache Hadoop Project: http://hadoop.apache.org.

[3] Tom White, Hadoop: The Definitive Guide, O'Reilly Media, June 2009.

[4] Jason Venner, Pro Hadoop, APress Publications, ISBN No: 978-1-4302-1942-2, June 2009.

[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, "The Hadoop Distributed File System", Mass Storage Systems and Technologies (MSST), IEEE 2^{6th} Symposium, 2010.

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

[7] The Next Generation of Apache Hadoop MapReduce, http://developer.yahoo.com/blogs /hadoop/posts/2011/02/mapreduce-nextgen/.

[8] www.ibm.com/developerworks/library/wa-introhdfs/.

[9] Konstantin V. Shvachko, "HDFS scalability: the limits to growth", usenix Vol. 35, No.3,May2010,www.usenix.org/publications/login/2010-4/openpdfs/shvachko.pdf.

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

[11] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, Ying Li, "Hadoop High Availability through Metadata Replication", IBM China Research Laboratory, ACM, November 2009.

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

[13] www.en.wikipedia.org/wiki/Distributed_hash_table.

[14] M.H. Braunisch, "A Thesis in the Field of Information Technology," Master of Liberal Arts thesis, Harvard Univ., 2006.

[15] G. S. Manku, M. Bawa, and P. Raghavan, Symphony: Distributed Hashing in a Small World. Appears in Proc. 4th USENIX Symposium on Internet Technologies

andSystems(USITS),pages12740,March2003.http://wwwdb.stanford.edu/~manku/papers/03usits-symphony.pdf>

[16] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, “Chord: A peer-to-peer lookup service for Internet applications,presented at the ACM SIGCOMM, San Diego, CA, Sep. 2001.

[17] A. Goel, R. Govindan, and H. Zhang. (2004) Improving lookup latency in distributed hash table systems using random sampling. Comput. Sci. Dept., Univ. ofSouthernCalifornia. http://www.cs.usc.edu/Research/TechReports/04-825.zip

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, “Dynamo: Amazon’s Highly Available Key-value Store”, in ACM symposium on Operating systems principles, Volume 41, Issue 6, Pages 205-220, December 2007

[19] Zhou, Shuheng; Ganger, Gregory R.; and Steenkiste, Peter Alfons., "Balancing locality and randomness in DHTs" (2003).ComputerScience Department.Paper 2238, http://repository.cmu.edu/compsci/2238.

[20] M. Elena Renda, Giovanni Resta, and Paolo Santi, “Load Balancing Hashing in Geographic Hash Tables”, IEEE Transactions On Parallel And Distributed Systems, Vol. 23, No. 8, August 2012.

[21] Sylvia Ratnasamy and Brad Karp, Li Yin and Fang Yu, Ramesh Govindan,Scott Shenker ,“ GHT: A Geographic Hash Table for Data- Centric Storage” in proceedings of the 1st ACM international workshop on Wireless sensor networks and applications, pages 78-87 , September 28, 2002 .

[22] Gurmeet Singh Manku, Moni Naor and Udi Wieder, “Know The Neighbor's Neighbor: The Role of Lookahead in Randomized P2P Networks, Proc. 36th ACM Symposium on Theory of Computing (STOC 2004),pages 54-63, June 2004.

List of Publications

Published

[1] Ravneet Kaur, Shalini Batra, "A Short Survey of Key Distribution Techniques for Namespace Load Balancing", International Journal of Emerging Technologies in Computational and Applied Sciences, 4 (5), March-May 2013, Pages 532-535.

Communicated

[1] Ravneet Kaur, Shalini Batra, "An Approach to Geographic Based Namespace Load Distribution using Symphony", International Journal of Computer Information Systems ISSN : 2229-5208.
(Communicated)