P2PHDFS: AN IMPLEMENTATION OF STATISTIC MULTIPLEXED COMPUTING ARCHITECTURE IN HADOOP FILE SYSTEM

A Thesis

Submitted to

The Temple University Graduate Board

In Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

By

Aakash Pradeep

January 2013

Thesis Approvals:

Dr. Justin Y. Shi, Thesis Advisor, Department of Computer and Information Sciences

© Copyright 2013

Aakash Pradeep. All Rights Reserved

ABSTRACT

The Peer to Peer Hadoop Distributed File System (P2PHDFS) is designed to store and process extremely large-scale data sets reliably. This is a first attempt implementation of the Statistic Multiplexed Computing Architecture concept proposed by Dr. Shi for the existing Hadoop File System (HDFS) to eliminate all single point failures. Unlike HDFS, in P2PHDFS every node is designed to be equal and behaves as a file system server as well as slave, which enable it to attain higher performance and higher reliability at the same time as the infrastructure up scales. Due to the data intensive nature, a full implementation of P2PHDFS must address CAP Theorem challenges. This MS project is only intended as the ground breaking point using only sequential replication at this time.

ACKNOWLEDGEMENTS

There are many people who have made the pursuit of my Masters degree possible. I would first like to thank my parents, my elder brother and sister and my wife for giving me the opportunity to achieve my goals and having stood by me in every decision I have made.

I would like to thank all the people involved in this project through their constant guidance, help and support. I would like to express my gratitude towards my advisor, Dr. Justin Y. Shi for his valuable guidance and encouragement to achieve success in my career.

Finally, we would like to thank our teammates, Moussa Taifi, Yasin Celik and Vivek Yeddanapalli for collaboration and discussions that are indispensable for making P2PHDFS to its current shape.

TABLE OF CONTENTS

Page

ABSTRACT III

ACKNOWLEDGEMENTS IV

LIST OF FIGURES VII

CHAPTER

1. INTRODUCTION 1

2. PROBLEM DEFINITION AND OBJECTIVES 5

3. PROPOSED STATISTIC MULTIPLEXED HDFS ARCHITECTURE 8

Ring Structure 8
Component Level Design 10

Data Node 11
Name Node 11
Node Manager 12
Tuple Receiver 12
Tuple Transmitter 13
Tuple Bucket 13

File system name space 13
Data Replication 13
Data Organization 14

File creation 14
File reading 15

Data storage reliability 16

Node heartbeat 16

Data block rebalancing 17
Temporary Namenode 17
Space Discovery Protocol 18
Block Discovery Protocol 19
4. IMPLEMENTATION 20
Implementation of Tuple 20
Structure of Tuple 20
Implementation of Tuple Bucket 23
Implementation of UVR 23
Protocol for Joining Node to UVR 23
Effects of node failure in different phase of Joining 24
5. SUMMARY AND FUTURE WORK 28
Summary 28
Future Work 30
BIBLIOGRAPHY 31

LIST OF FIGURES

Figure 1: UVR of 2-Node 8

Figure 2. UVR of 3-Nodes........ 9

Figure 3: Component Level Design........10

Figure 4: File Creation 15

CHAPTER 1

INTRODUCTION

Hadoop File System (HDFS) [1] is a subproject of the Apache Software Foundation project Hadoop. It is a system designed for storing and processing large amounts of data, in terabytes and petabytes. HDFS allows to connect nodes (commodity personal computers) contained within clusters over which data files are distributed. One can then access and store the data files as one seamless file system. But, internally file is divided in fix size of small chunks called block except the last block, all other blocks are of same size and they are stored on slave nodes called Datanode. Blocks are replicated on different Datanode to make HDFS fault-tolerant. In HDFS, all metadata are stored on a single node called Namenode, which roughly uses around 64 byte for storing metadata for each block. Namenode is the master node, it is responsible for deciding which Datanode will store which block and also have an in-memory mapping for file to its block and block to Datanodes, where it is stored. Namenode is also responsible for maintaining replication for each block. This centralized controlling architecture makes HDFS easy to maintain and enable it to overcome any number of Datanode failure but at the same time it makes it a single point failure system, as it loses all metadata if Namenode fails. Though they have the secondary name node concept but it involves manual intervention and minimum downtime is 30 minutes. Since Namenode is the only node, which stores the metadata information, and all the metadata is stored in-memory, the Namenode memory becomes limiting factor for the number of files an HDFS can store.

The P2PHDFS has been designed to address above-mentioned shortcoming of HDFS. In P2PHDFS we store the metadata in distributed fashion. Instead of designating a node as Namenode and store all the metadata in it, P2PHDFS stores both metadata and data at each node. That is instead of having one server and many slaves architecture of HDFS, in P2PHDFS all nodes are equal and every node is server as well as slave. This virtue of P2PHDFS makes it a highly scalable system and enables it to overcome the single point failure problem of HDFS. In addition, since there is no single Namenode, where all the metadata is stored, P2PHDFS does not have any limit on the number of files it can store. This concept also helps in attaining high fault-tolerance; it is designed to tolerate failure of any node. Whereas in other distributed file systems like HDFS, PVFS [2], Lustre [3] and Google File System (GFS) [4] the metadata and application data are stored separately but in P2PHDFS a node is responsible for storing the data as well as metadata. The nodes in P2PHDFS do not use data protection mechanisms such as RAID to make the data durable. Instead, the file is divided in chunks of equal size except the last chunk, which may or may not be of same size, and each chunk is replicated on separate nodes for reliability and fault tolerant purpose, which is same as in current HDFS. The number of replication for a chunk is a file level property that is defined by the user while creating the file, by default it is 3.

P2PHDFS files system assumes that file will be written once and read many times and in the current version it does not support appending, but expected that it will support appending as well as modification of the file in the coming version. P2PHDFS also assumes that file will be read in sequential way and one block at a time. While designing P2PHDFS, it has been taken care that it supports all the HDFS API, so that it is compatible with MapReduce framework [7]. Therefore, the user needs not to do any API change in the client code; only some configuration file change is required.

As discussed earlier, there is no single server node in P2PHDFS, so when a client wants to do any file operation, it contacts one of the node, which is supposed to be picked up randomly, so that the load is distributed and that selected node of P2PHDFS becomes temporary Namenode for that client. In P2PHDFS, every node maintains an optimized list of nodes based on rack-aware policy and it also maintains mapping of the blocks to Datanodes as done in HDFS but only for the blocks stored on the current node. In this system, every block stored in Namenode of P2PHDFS, has its associated metadata to identify which file it belongs to and its index number to that file. These data structures enable a node to support writing of a file and also enable it to temporarily store the metadata of a file and it's blocks to support the read. If it does not have metadata to support read for a particular file then it will ask other node to provide that information and after receiving that information, it reconstructs the data structures and allows the client to read the file seamlessly. Since P2PHDFS assume that the file is read in a sequential way, instead of asking particular block information, it just generates a request to read next 10 block's metadata. It helps the system, by not letting it overload the network with metadata and also fasten the data collection and reconstruction of data structures. But the block storage is still same as in Datanode of HDFS.

Unidirectional Virtual Ring (UVR) is backbone of P2PHDFS. All nodes of P2PHDFS are part of this UV Ring. For any node to be part of P2PHDFS needs to join this UVR. Each node has been assigned a unique ring id in UVR. And every node's successor node is responsible to inform the other nodes in the UVR about its failure. In order to do so, every node sends heartbeat to its successor node. Every node can also communicate to other nodes using the Tuple Space System, which is different in nature than one proposed by the David Gelernter [5]. Here, tuple is used to communicate commands instead of data. But most of the times these command are generic in nature that is, it is not addressed to any particular node and whichever node thinks it has answer will return the information to the concern node. For example, if a node wants to have metadata for a file, it creates a tuple command and sends it to its successor node which transfers tuple to its successor node and so on and whichever node has the information regarding that file will send information back to the requesting node. Every node has a component to receive the tuple and communicate it to different components through tuple buckets and take decision whether that tuple need to be further transmitted or not based on the hop count property of tuple. Hence, in P2PHDFS every node has a tuple space used by the node itself and at the same time it is also a part of global command tuple space of UVR.

CHAPTER 2

PROBLEM DEFINITION AND OBJECTIVES

Hadoop Distributed File System is designed to store large file, here large file means file of size of several gigabytes and terabytes, in efficient and fault tolerant way at the same time the emphasis is on high throughput read access rather than low latency of data access. It treats node failure as expected event rather than exception. And in order to make system fault- tolerant and to create a system of commodity hardware, HDFS follows one master and multiple slave architecture. In HDFS, they have one server node called Namenode and many slave nodes called Datanode. It stores all the metadata at Namenode and all the file data is stored at Datanode. In this system, each file is divided in small chunks called block. Block's metadata is stored at Namenode and block’s data is stored at Datanode. Datanode treat each block as a file and it is not aware of mapping of files to block and vice versa. Though this enables this system to overcome any number of failure in Datanode and also makes decision making easy for Namenode, since it has all the metadata. So Namenode can take decision, to store that block on which nodes and maintains the replication of blocks across the system. Even client talks to these Namenode and gets all the required information like list of Datanodes where a particular block’s data is stored and new list of Datanodes where block’s data to be stored. But this architecture also makes it a single point of failure system. If Namenode fails, all metadata is lost no read and write operation can be performed and information stored at Datanode is not good enough to recreate the metadata. To overcome, this problem in HDFS checkpoint concept has been implemented which on fix interval writes metadata to a file and but it does not write everything, it just stored the file to blocks mapping. Secondary Namenode, keeps reading these metadata file periodically. But when Namenode fails, to make Secondary Namenode as primary requires manual intervention as well as its initial status is safe mode, where it would not perform any file operation like reading and writing. In this safe mode, it keeps receiving the block report from different Datanodes,block report informs that this Datanode has these many blocks and Namenode try to reconstruct the required data structures. This whole operation take a while, minimum expected time is 30 mins and it depends on the size of cluster, if it is very big then this operation may even taken longer time than this. The point to be noted down here is that since Namenode is not active, no file operation can be performed and so even MapReduce framework would not work. Hence the whole system will be inactive in that time period. Besides this, since all metadata is stored at Namenode only and every metadata for a block requires 60 bytes, hence random access memory of Namenode becomes the limiting factor for the number of files it can store. This puts a limiting factor on HDFS and defies its horizontal scalability feature beyond a certain limit.

P2PHDFS address these problems of single point of failure and horizontal scalability issue while maintaining the transparency to the client of all the internal details as well as fault tolerant in such a way that it can tolerate failure of any node in the system. It is also suppose to support same API as HDFS API, so that user need not to change the client code for reading and writing file, rather it is expected that only a scheme change is required in configuration file for file system. It is also expected that P2PHDFS will have same high throughput of data access.

P2PHDFS is designed to be a new peer to peer distributed file system for the Hadoop framework. It is based on a uni-directional virtual ring and command tuple space concept. P2PHDFS is designed to have high performance and scalability.

The objective of P2PHDFS is,

- To build a reliable distributed file system with automated replication and has a high availability.
- To gain high performance and reliability as the system and infrastructure gets bigger.
- To have zero single point of failure.

CHAPTER 3

PROPOSED STATISTIC MULTIPLEXED HDFS ARCHITECTURE

P2PHDFS System creates a unidirectional virtual ring (UVR) Structure of nodes and creates a tuple space for communication between different nodes. In this UVR each node is identified by a Ring-Id, a hash value of its Datacenter-Id/Rack-Id/IP and supposed to keep a routing table of size m, where m satisfy the following equation: $N <= \min (2m))$, here N is the total number of nodes in the system. There are two kind of routing tables maintained by each node, one routing table keeps hostname and ring id of m successor node and other routing table maintains hostname and ring Id of m predecessor node [3].

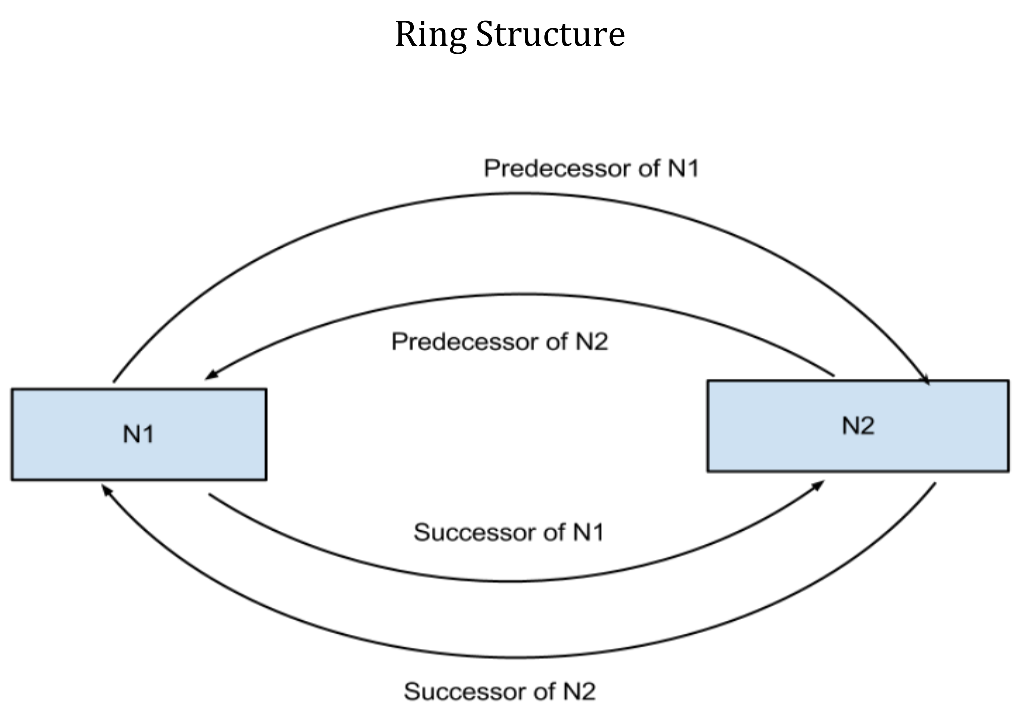

FIGURE 1: UVR OF 2-NODE

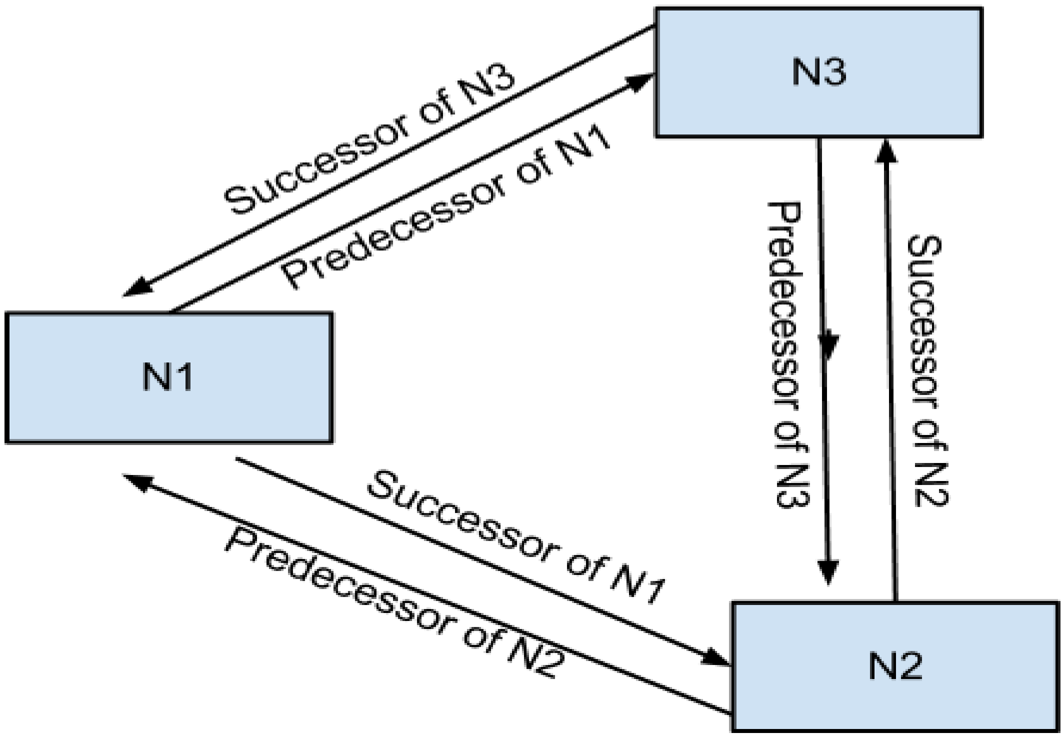

FIGURE 2: UVR OF 3-NODES

In the UVR of nodes of P2PHDFS, each node will have one predecessor and one successor node and will also have a list of m-1 predecessor and a list of m-1 successor nodes. So, for example, if we take 2 node UV ring then N=2 and m=1 hence for node N1, successor would be N2 and predecessor will be also N2 and since m=1 it would not have any successor node list and predecessor node list. Same for Node N2, N1 will be successor as well as predecessor and will have empty successor node list and predecessor node list. Whereas in case of UV ring of 3 nodes, N=3 and so m=2 and N1 will have, N2 as successor and N3 as predecessor and since m=2, hence it successor node list will have

N3 (since N3 is successor of N2) and will have N2 in predecessor node list (since N2 is predecessor of N3).

Component Level Design

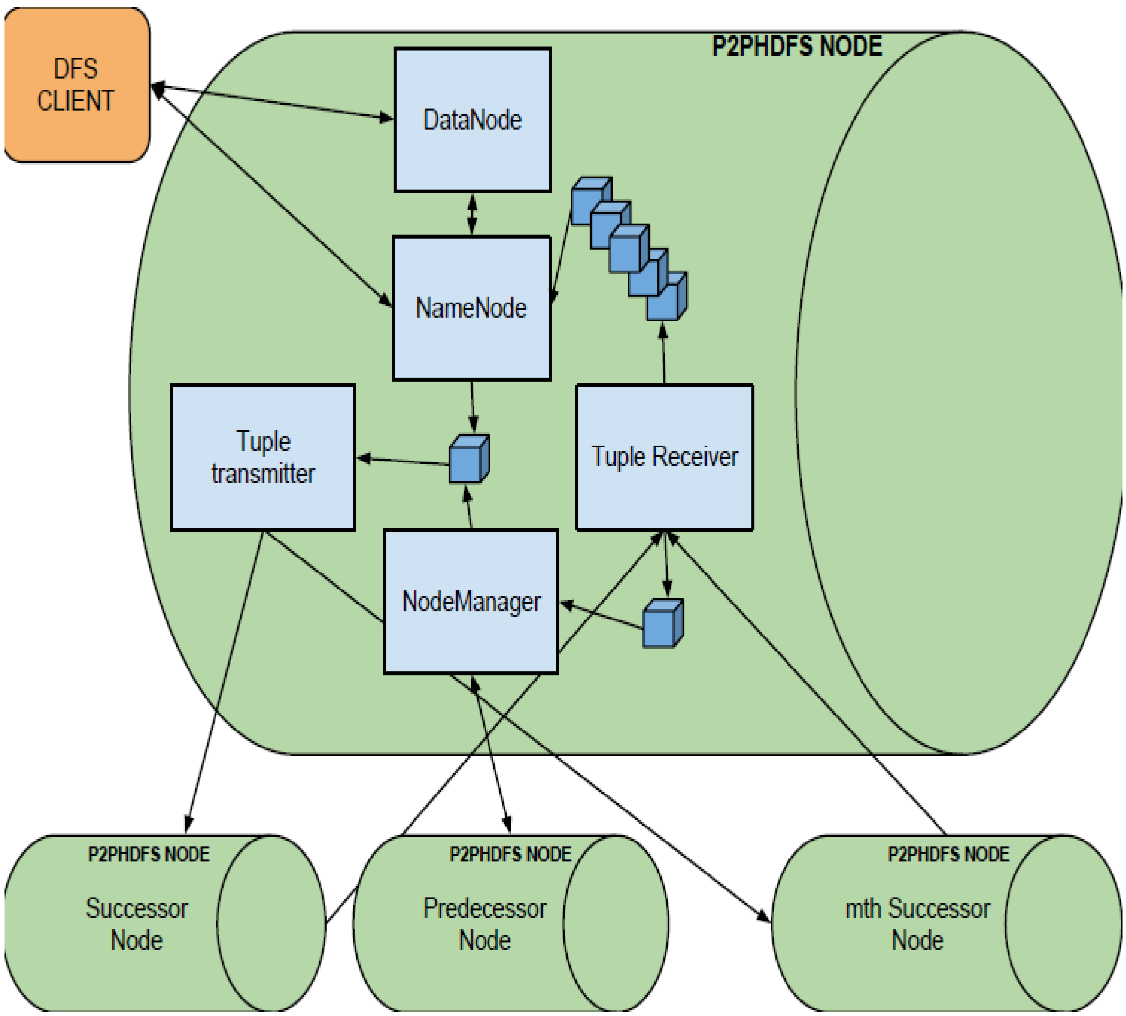

FIGURE 3: COMPONENT LEVEL DESIGN

Each node in the UVR of P2PHDFS will have following components: -

1 Data Node

2 Name Node

3 Node Manager

4 Tuple Transmitter

5 Tuple Receiver

6 Tuple Buckets

Data Node

Each node will have one Data Node component that is responsible for storing the blocks efficiently and for managing them. On request, it will transfer block data to client and to other data node and will also receive block's data from client and from other data node to store them.

Name Node

Name node is responsible for managing the metadata regarding the blocks and files. It stores the mapping of files to block and also stores the map of which block belong to which file. It also maintains which block is stored on which data nodes. It has following other responsibilities:-

a Commanding data node to perform particular job.

b Maintain replication of blocks.

c Allocating new blocks for a file and deciding which data nodes will store it.

d Maintaining lease for client.

e Deleting some bad blocks, as reported by client

f Keep sending Disk space information to other nodes using Tuple

g Manage the list of alive data nodes

h Decommission a particular node and replicate all blocks stored on that node.

Node Manager

Node manager is responsible for maintaining the UVR and routing table. Every node manager will receive heartbeat from the predecessor's node manager and if it does not receive heartbeat for a particular interval then assumes that predecessor node is failed and issue a DECOMMISSION tuple.

Tuple Receiver

This component is responsible for receiving the tuple from other nodes and based on the COMMAND type it stores them in a particular tuple bucket. Other components keep waiting for the tuples in tuple bucket. On receiving tuple, it also takes decision whether it needs to be transmitted further or not based on its hop count. By default, each tuple has hop count value as 3. Hop count is set to 3 if m is greater than or equal to 3 otherwise it is set to m.

Tuple Transmitter

This component is responsible for transmitting the tuples to other nodes based on it's destination type. If destination type is NORMAL then it will transmit the tuple to immediate successor node and m^{th} successor node. Before transmitting, it decrements the hop count value by 1.

Tuple Bucket

Tuple bucket is responsible storing the tuple in first come first serve order. It has API to let the client wait for particular amount of time for a tuple or client can also block itself till the tuple is available.

File system name space

P2PHDFS supports a traditional hierarchical file organization in which a user or a application can create directories and store files inside them. The file system namespace hierarchy is similar to most other existing file systems; you can create.

Data Replication

P2PHDFS 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 cannot be changed any time after that. The name node makes all decisions concerning block replication. P2PHDFS uses an intelligent replica placement model for reliability and performance.

Optimizing replica placement makes P2PHDFS unique from most other distributed file systems, and is facilitated by a rack-aware replica placement policy that uses network bandwidth efficiently. Large P2PHDFS environments typically operate across multiple installations of computers. Communication between two data nodes in different installations is typically slower than data nodes within the same installation. Therefore, the name node attempts to optimize communications between data nodes. The name node identifies the location of data nodes by their rack IDs.

Data Organization

One of the main goals of P2PHDFS is to support large files. By default the size of a typical P2PHDFS block is 64MB but it can be specified by the client while creating the file and this value cannot be change later on. P2PHDFS tries to place each block on separate data nodes. There are three important steps in data organization.

File creation

P2PHDFS allows file creating using the DFS-Client and the APIs are same as HDFS. Once file create, it cannot be change. All the block creation and replica management of blocks and transferring of data to a particular data node remain transparent to client; DFS-Client internally takes care of it. For client, it is just like writing normal file using output stream in Java Programming language. When an application writes a file, DFS-Client first asks the name node for the list of data nodes that can host replica of the block of file. The list is sorted by the distance from the client. DFS-Client first transfers the data from client to the first data node, which on receiving transfer the block to the next data node in the pipeline and the next data node also does the same. After receiving the blocks, data nodes inform its Namenode about it, which update its metadata. The interaction between client, name node and data node is illustrated in following diagram:

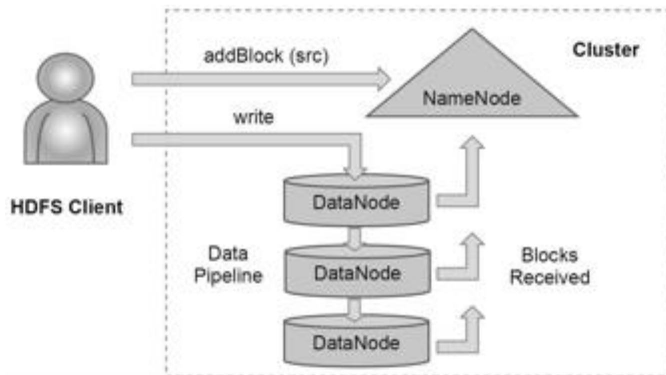

FIGURE 4: FILE CREATION

File reading

P2PHDFS allows file reading using the DFS-Client and the APIs are same as HDFS. For reading a file, client need to specify the file path and this path should be absolute. On receiving the path the Namenode looks into its metadata to find the list of Datanodes where blocks are stored, if it does not find the file and its requested block then, it asks from other nodes. On receiving their response, it reconstructs the data structure and returns a sorted list of Datanodes to client.

Data storage reliability

One important objective of P2PHDFS is to store data reliably, even when failures occur within name nodes, data nodes, or network partitions. Detection is the first step P2PHDFS takes to overcome failures. P2PHDFS uses heartbeat messages to detect connectivity between nodes.

Node heartbeat

Several things can cause loss of connectivity between nodes. Therefore, each node manager sends periodic heartbeat messages to its successor node manager, so the latter can detect loss of connectivity if it stops receiving them. The node manager marks a predecessor node as dead node when it does not receive heartbeat from it for a configured time period and sends a decommission tuple for that node. Since the node is failed, hence data blocks stored on that node is no longer available and so other P2PHDFS node need to update their metadata for blocks which were stored on that data 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.

Data block rebalancing

P2PHDFS data blocks might not always be placed uniformly across data nodes, meaning that the used space for one or more data nodes can be underutilized. Therefore, P2PHDFS supports rebalancing data blocks by moving data blocks from one data node to another automatically if the free space on a data node falls too low. Another model might dynamically create additional replicas and rebalance other data blocks in a cluster if a sudden increase in demand for a given file occurs.

One common reason to rebalance is the addition of new data nodes to a cluster. When placing new blocks, name nodes consider various parameters before choosing the data nodes to receive them. Some of the considerations are:

- Block-replica writing policies
- Prevention of data loss due to installation or rack failure
- Reduction of cross-installation network I/O
- Uniform data spread across data nodes in a cluster

Temporary Namenode

When a client wants to perform file operation, it sends its request to an active node in P2PHDFS and that node becomes temporary Namenode for that client. Since all nodes are equal in P2PHDFS, hence any node can become Temporary Namenode for any client. Once a node becomes a temporary Namenode for a client, client sendsall its request to that node only. If that node fails it again associate itself with some other node in P2PHDFS and now that node will be temporary Namenode for that client and will serve all the requests. It is called temporary Namenode, since unlike HDFS where a node is always designated as Namenode and all other nodes are slave and whole lifetime of that node it will be Namenode for all the client, but in P2PHDFS any node can be Namenode for any client till the time period that client is requesting. Once the client releases the lease it may become a Namenode for any other client or may not serve any client. So, a node behaves as a Namenode till he serves the client, therefore its status of a Namenode is temporary.

Space Discovery Protocol

In order to make decisions of selecting data nodes where a block can be written, a Namenode of a node should have sufficient information about available disk space, total disk space capacity and number of processes running at that Datanode. This metadata information for a Datanode should be available as per a policy, which should be rack and datacenter aware, so that blocks are stored in way that load is balanced across the P2PHDFS. In order to have this information at every node, every node's Namenode sends a tuple called SPACE_TUPLE, having available disk space; total disk capacity and number of processes running information, at a particular interval of time and all the nodes, which receive this information, update their metadata. Besides this, when a node join the UVR it sends a SPACE_QUERY tuple, and on receiving this tuple old nodes send the SPACE_TUPLE to the entire system. Till a newly joined node has sufficient information about Datanodes, it remains in inactive state. Once it gets sufficient information about Datanodes, it changes it state to active and become ready to serve a client as Temporary Namenode as well as can be used as Datanode by other node. After being in active state it sends a SPACE_TUPLE to the entire system order to make them aware of its space availability.

Block Discovery Protocol

When a node behaves as a Temporary Namenode for a client and it gets a read request for a file's block, for which it does not have the required metadata, current node issues a BLOCK_DISCOVERY tuple to the P2PHDFS and it waits for the metadata information from other node for a particular amount of time. On receiving the metadata information for that block it respond back to the client as well as request for next 10 blocks of that file i.e. it issues a BLOCK_DISCOVERY tuple asking for next 10 blocks, as files are read in sequential way, hence it is highly expected that client will ask for the next block also.

In order to reply for this tuple, every node should store some extra metadata with each block. Every block's metadata should have information about the file it belongs, the starting byte in context of file, size of the block and its index in file. This information will make the entire system stateless and enables any node to reconstruct the data structures required to serve a file for reading purpose, and therefore any node in P2PHDFS can be assigned as Temporary Namenode to any client.

CHAPTER 4

IMPLEMENTATION

P2PHDFS is implemented in Java and one of design goal is to support same APIs, as by HDFS, in order to make, it is easier to switch to P2PHDFS from HDFS. It is designed in such a way that for using the P2PHDFS, client just need to change the scheme of Namenode's url in their core.xml file and use the APIs in same way as they use for HDFS for file operations.

Implementation of Tuple

Structure of Tuple

ID	Tuple Command	Source NodeIdentifier
Destination Type	Destination NodeIdentifier	Hop Count
Payload Existence	Payload	

ID

It is an UUID Number and an object UUID class provided in JDK. And according to Java documentation it represents an immutable universally unique identifier. A uuid represents a 128-bit value. It makes a Tuple unique across, the global tuple space.

Tuple Command

This is implemented as Java Enum and it represents the command for which tuple is created. It can be for reading the file or for space discovery.

Source NodeIdentifier

This represents the source node which has created the Tuple. It is an object of NodeIdentifier class, which stores the unique ring id, hostname and port.

Destination Type

This represent the type of destination tuple will be transmitted. It is again implemented as Java Enum and it can have only two types of value UNIQUE or NORMAL. If it is UNIQUE, this tuple is supposed to be transmitted to a single designated host only, this is to be used rarely. Other one is NORMAL; by default this is the destination type for a tuple. This type specifies that tuple will be transmitted in normal way that is to immediate successor node and to m^{th} successor node.

Destination NodeIdentifier

There would be a value for Destination NodeIdentifier; if the Destination Type is UNIQUE otherwise it will have NULL value. It is an object of NodeIdentifier class, for destination node.

Hop Count

This represents the number of times it should be transmitted and indirectly it also specifies the lifespan of a tuple. It is a short integer value and by default it is 3. And after each transmission, tuple copied from one node to another, it will be reduced by 1. And node will keep it transmitted till it is 0. For Tuple having Destination Type as UNIQUE it is set to 1.

Payload Existence

It specifies whether tuple has any payload. It is defined as Java Enum and it has only two types of value ABSENT and PRESENT.

Payload

It represents data that tuple carry, if its Payload Existence is PRESENT otherwise it will be NULL.

Implementation of Tuple Bucket

A tuple bucket is storage, where tuples are stored. It provides APIs to read and write tuple from the bucket. Each tuple bucket has a field bucket type, which specifies the type of the tuple bucket. At any point of time there can be only one tuple bucket of a particular type. Tuple bucket stores tuple in a queue and at any point of time only one thread can read from it and only one thread can write tuple to it, that is at the same time one thread can write and other thread can read but no more than one thread can perform read and write operation at the same time. In addition it provides API to make a blocking calls that is, if we want to read a tuple and if it is not available then the thread will wait for a given time and same kind of API exist for writing also, but since in the current implementation there is no limit for the number of tuple that can be stored in a tuple bucket, hence it never became blocking in nature that is the thread writing a tuple to tuple bucket will not wait.

Implementation of UVR

Protocol for Joining Node to UVR

Step 1. JOIN

The node wants to join sends a JOIN Tuple to any node in UVR, which will further transmit this tuple in normal way to other nodes in UVR.

Step 2. JOIN_OK

Future successor node will send JOIN_OK tuple to requesting node and in this tuple it will send its finger table that will be the finger table for the requesting node. Upon receiving which the requesting node will update their finger table and start sending the heartbeat to successor node.

Step 3. JOIN_COMMIT

Current node sends a JOIN_COMMIT to successor node and mth successor node. And all other node on receiving this tuple adjust their finger table and predecessor node of the node joining the UVR will start sending heartbeat to it.

Effects of node failure in different phase of Joining

JOIN

The node want to join sends a JOIN Tuple.

a Current node fails

i If it fails before sending the JOIN tuple, there is no effect and no changes required.

ii If it dies after sending the JOIN tuple then the node that decides that it will be its predecessor will send JOIN_OK tuple and will expect heartbeat from the node requested to join UVR and at the same time expect the heartbeat from earlier predecessor. If it does not receive the heartbeat from the predecessors node for a given interval then it will send the DECOMMISSION tuple for it.

b. If future successor node fails before responding JOIN_OK signal then the requesting node will wait for a given time to get the JOIN_OK tuple and on not receiving it, it will resend the JOIN tuple. In current implementation the node will retry 3 times to join the UVR.

JOIN_OK

Future successor node will send JOIN_OK tuple to requesting node and in this tuple; it will send its finger table that will be the finger table for the requesting node.

a Node want to be in UVR fails

i On not receiving heartbeat for a given interval successor node will send a DECOMMISSION Tuple.

b Future successor node fails:

i On receiving the JOIN_OK tuple requesting node will send the JOIN_COMMIT signal to the current successor node and to mth successor node and hence to all other nodes. And on receiving the DECOMMISSION tuple of its successor node it start to contact the next successor node.

c If predecessor node fails:

i Since it received the JOIN_OK tuple, hence it expect heartbeat from the predecessor node and sends the JOIN_COMMIT tuple to make the predecessor node aware of its presence. But before receiving the JOIN_COMMIT tuple predecessor node will keep sending the heartbeat to its successor node and if it fails then it is responsibility of that successor node to send the DECOMMISSION tuple for it.

JOIN_COMMIT

a current node die

i Successor node will update its finger table that it will have only one predecessor node on not receiving the heartbeat from it and sends DECOMMISSION tuple.

b successor node fails

i In normal case JOIN_COMMIT will send before DECOMMISSION tuple issued for its successor node, so it will make aware other nodes of UVR about its presence. At the same time when a predecessor node sends heartbeat it expects its successor nodeIdentifier in response. On not getting its response for a given time, it will try to connect to the next successor node in its Finger table and if it receives the nodeIdentifier of this new successor node then it updates its finger table by requesting the new successor node for it.

c predecessor node fails:

i It will not receive the heartbeat for a given time interval and will send a DECOMMISSION tuple to all other node for it.

CHAPTER 5

SUMMARY AND FUTURE WORK

Summary

This project is the very first attempt applying the Statistic Multiplexed Computing concept for a very large scale file system. We have designed a “pure” peer-to-peer architecture and protocol for P2PHDFS as a foundation for future full implementation. Using Statistic Multiplexed Computing, we aim to overcome the scalability inhibitors in the Hadoop Distributed File System. These include single point failures and limitation in number of files stored in HDFS. In the proposed prototype system, there is no single master server controlling and monitoring the entire file system, instead the metadata is stored in fully distributed fashion which enables the system to treat all the nodes equally; hence no single node failure can bring down the entire system. This also enables the file system to scale horizontally to any physically possible limit.

By introducing the concept of Temporary Namenode, P2PHDFS also enable every node to behave as a master for a client, who eventually helps in distributing the load of serving client to all the nodes rather than loading only one master server as in HDFS, we are hoping that this will improve the client serving power of system. We also change the metadata stored with each block to enable the system to recreate the entire metadata for a file and allow any node to create the mapping of a file to blocks. This introduction of extra metadata per block helps the system in overcoming any node failure, since the metadata for a file that has been stored on the failed node can be created again at any other node even if this node fails. Therefore, it can be stated that the system is stateless. As for reading a particular file on a client request, the P2PHDFS need not to find out the node which has metadata for the requested file to fulfill the client request.

The UVR structure of the system, the heartbeat mechanism between a node and its successor and routing information of finger table makes P2PHDFS system very robust and allow it to overcome any number of nodes failure. It makes P2PHDFS capable to add any number of nodes in the ring. The most powerful future of the system is automatic rearrangement of successor and predecessor node and automatic update to Finger table on addition and failure of any node.

P2PHDFS system also uses the concept of tuple space but it uses for transmitting the command and information in tuples in a very distributed fashion, since none of tuples is addressing any single node. In fact, it has been transmitted to a global tuple space which consist of the private tuple space of each node and so any node which needs can consume the information as well perform the operation and contribute back the results. The use of tuple only to transmit the commands helps in reducing the network load since in the P2PHDFS file system, it has always been preferred to move computation rather than data since we are dealing with the file of gigabytes and terabytes in size.

The proposed P2PHDFS system is a promising technology to read and write file of gigabytes and terabytes with high throughput of data access without scalability limits.

Future Work

In order to make it consumable and ready for the industry we need to test it more on different platform and on different scale as currently it has been tested only on small cluster of size 2-10 in TCloud environment. Till now, we have just tested the functionality of the system, but in future we would like to measure the performance of the system on different cluster size and need to find out the places where we can make this system more efficient. We also need to measure how much time a node takes to join a ring on large cluster and we need to find out, how we can reduce the amount of data transferred while joining the node in order to reduce the load on the network. Same way we need to improve the network uses and reduce the amount of data transfer to rearrange the ring when a node fails. Since no system is perfect, we still need to find out more practical as well as ideal scenarios where this system will fail and we also need to fix them and we need to make it more efficient, scalable and robust distributed file system. On reaching to a stable state we would like to extend this system to perform update and append operations to a file.

BIBLIOGRAPHY

1. DhrubaBorthakur, The Hadoop Distributed File System: Architecture and Desgin, http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf

2. Philip H. Carns, Walter B. Ligon III, Robert B. Ross and Rajeev Thakur, PVFS: A Parallel File System for Linux Clusters,

3. Sun Microsystem, Lustre File System, http://www.csee.ogi.edu/~zak/cs506-pslc/lustrefilesystem.pdf

4. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System, http://www.cs.rochester.edu/meetings/sosp2003/papers/p125-ghemawat.pdf

5. David Gelernter, "Generative communication in Linda" ACM Transactions on Programming Languages and Systems, volume 7, number 1, January 1985

6. Ion Stoica, Robert Morris, David Karger, M. FransKaashoek, HariBalakrishnan, Chord: A scalable Peer-to-Peer Lookup Service for Internet Applications, MIT Laboratory for Computer Science, http://pdos.lcs.mit.edu/chord/

7. Jeffrey Dean and Sanjay Ghemawat, Google, MapReduce: Simplified Data Processing on Large Clusters

8. Justin Y. Shi, “Statistic Multiplexed Computing Architectures,” Technical Report, CIS Department, Temple University, November 2012.

9. Justin Y. Shi, "Statistic Multiplexed Computing Architectures and Their Applications," Supercomputing 2012 Conference, Research Exhibit, Center for Networked Computing, Temple University.

10. Justin Y. Shi, Suntiang Song, "On the Assumptions of CAP Theorem and Big Data Challenges," International Journal on VLDB, in review, December 2012

11. Alexei Semenov, Evolution of Peer-to-Peer algorithm: Past, present and future, Helsinki University of Technology.