

# **A simulation of distributed file system**

Shan Tao, Xi Chen and Xiaoming Sun  
COEN 241: Cloud Computing, Summer 2014

# Table of Contents

1. Abstract.....	3
2. Introduction.....	3
3. Theoretical bases and literature review .....	5
4. Hypothesis.....	6
5. Methodology .....	7
6. Implementation .....	8
7. Data analysis and discussion.....	17
8. Conclusions and recommendations.....	28
9. Bibliography .....	29
10. Appendices .....	31

# Abstract

We have simulated SimpleFS, a simple and scalable distributed file system for nowadays highly demanded data parallel computations on clusters. To mimic the most popular concepts in distributed file system design, SimpleFS is comprised of three parts: client, metadata server and data server. Java socket is used to simulate the communications among the three. Command are inputted from the client terminal and sent to the metadata server for initial process. Information will be returned to the client directly. Operations like file I/O needs a second communication step between client and the data server. Java threads are used to mimic storage devices in the data server center. For writes, files are chopped into chunks and distributed across the storage devices. Data replications are also taken into consideration to protect the system under frequent device failure circumstances. Upon the simulation, files are able to be successfully stored into and retrieved from the system, even when storage device failure happens.

# Introduction

The goal of this study is to simulate a distributed scalable, reliable, high-performance file system to accommodate the on improving demand for parallel computations.

The performance of file systems has long proven to be critical to the overall performance of all kinds of computer applications. In order to find a better solution for the problem, various studies have been carried out in the past decade, among which Network File System (NFS) is the most traditional solution[1]. It allows one server to expose file system hierarchy and let clients map it into their local name space. Although widely used, the centralized server model makes the

whole system not scalable enough for today's extremely large amount of and dynamic data. Because of its centralization, the system is not reliable enough if device failure happens.

With cloud becoming more and more dominant technique for handling all kinds of computer applications, the amount of data become large enough that it has become very inefficient and energy costing to use one centralized server to do all the jobs like in the traditional way. There is a need for more scalable and reliable file system that does not rely on one single server, handles device failure properly, and gives high performance even when the volume of request form server is huge. Developing a suitable distributed file system has become crucial for the normal operation of cloud computing system.

The concept of distributed file system was thus introduced recently to fulfill the requirement of cloud computing. In cloud computing system, component failure is norm rather than the exception. File sizes are in TBs and is hard to handle using traditional method. Small pieces are desired. Most importantly, multiple servers have to be used to store the huge amount of data, thus producing many problems like synchronization and data distribution algorithm etc.

There has been many distributed file system model proposed to improve scalability. Google File System (GFS), after careful design and thorough testing, has been put into market and proven to be effective [2]. Other file systems, like TidyFS, ASDF, Ceph, etc, utilize similar concepts but take more factors into consideration, tending to improve the system performance even more. They all show improvement to different extent [3][4][5].

We are aiming to simulate a simple distributed file system with the most important features of other distributed file systems and the potential of implementing more complicated features. We are focusing on two types of commands the client might input: commands involve only the

metadata server (getFileSize, getDiskSize, rename) and those involve both metadata and data server (read, write). By using these most basic but typical commands, we are showing the effectiveness of nowadays emerging distributed file system models in handling cloud computing jobs.

## Theoretical bases and literature review

In recent years, shared-nothing compute clusters are a popular platform for scalable data-intensive computing. The typical frameworks, such as MapReduce [6], Hadoop [9], Dryad [10], are able to run data-parallel program which is possible to achieve very high aggregate I/O throughput.

Distributed file systems have been developed to support this style of write-once, high-throughput, parallel streaming data access. These include the Google File System (GFS) [8], and the Hadoop Distributed File System (HDFS) [11, 12]. Those systems have similar design:

- I. metadata for the entire file system is centralized and stores all information to describe the mapping and location, length of data, etc.
- II. data are stored separately on computing nodes, and replication is made by some algorithm.
- III. each node can finish task separately to use data stored on it, because data is not shared.

This paper presents SimpleFS, a distributed file system which is to simplify the system as far as possible by exploiting this restricted workload. It has the following properties:

- A. data are stored into different computing nodes, metadata describes the location, the length of the data, the mapping between stream and data, etc.
- B. each computing node can separately finish the computing bases on the data which is storing on it.

C. in order to simplify fault-tolerance and reduce communication the computing frameworks do not implement fine-grain transactions across processes

D. replication is lazy, which means the replication will be finished in near future, not immediately, which can be performed when the computing node is not busy.

However, there are some difference with popular DFS, such as GFS allows updates in the middle of existing streams, and concurrent appends by multiple writers, while the HDFS community has struggled with the tradeoff between the utility and complexity of even single-writer append operations [10] to a stream that can be concurrently read.

Our goal is to achieve scalability and fault-tolerance while remaining relatively simple. our simulation has the similar design as the popular distributed file system, which has client, data server and metadata server. Client talks to metadata and data server directly, read or write data with internal interfaces. Metadata server is very important which is applying an auto-replication algorithm to avoid server crash. The stream data cannot be read until the writing finish. And the update operation will provide a new copy for all data across the computing nodes instead of update the specified part of data, which can simplify the lock management. Lazy replication is another good point to balance the resource of computing nodes.

## **Hypothesis**

In this paper, we simulated a simplified distributed file system and achieved all the key functionalities of real DFS, including massive distributed storage, good scalability and fault tolerance, etc.

Based on this simulation, we could further research different metadata algorithms to compare their performance and load balance, test system availability of reacting to random server crashes, and improve the extensibility by adding more server nodes.

## Methodology

The architecture of this simulation is based on GFS. We created client, metadata server and chunk servers in the system. Client is the user to write and read files. Metadata server is the single master, managing the file directory and chunk servers are the physical storage for the data.

Each chunk server is simulated by one Java thread with pre-assigned storage capacity. Chunk servers hold the chunks of files and their replicas. It can also experience short term failure by killing the thread randomly.

Regarding the metadata server, it maintains namespace, mapping from files to chunks and their locations. We also used random and weighted random algorithm to assign the chunk location for different files, which can be further tested for load balance.

Client can write and read files in this simulated distributed file system. Additional services are also provided, including requesting storage current available capacity and name list for all the files in the system.

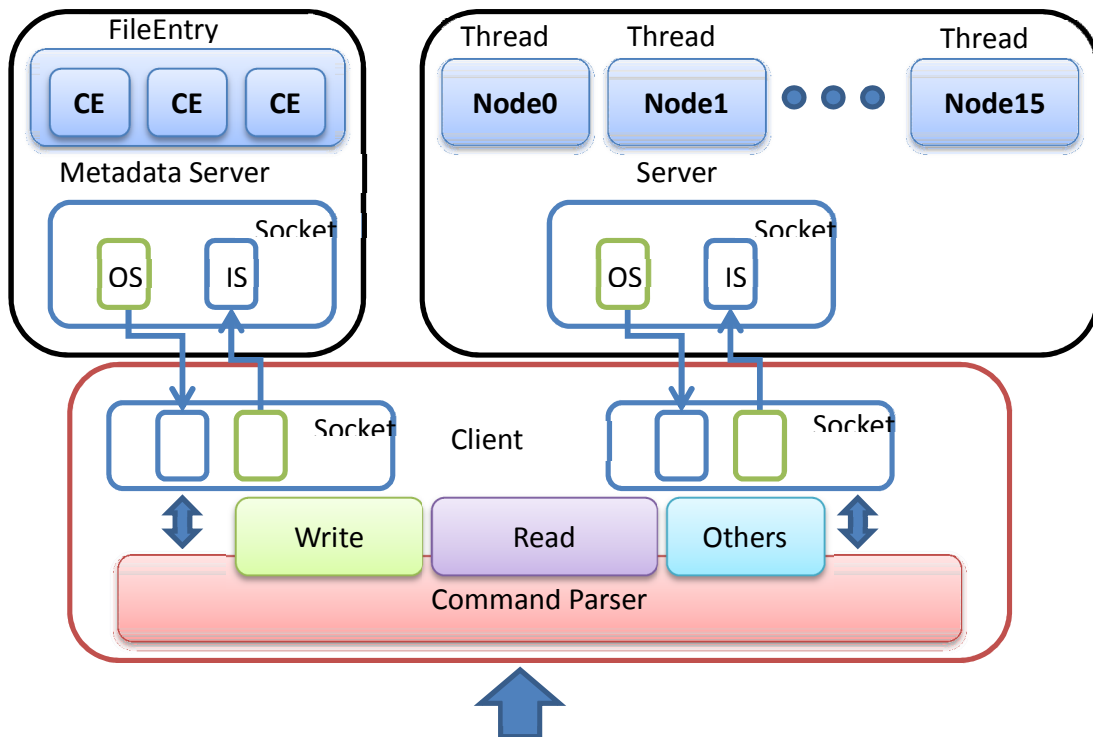
The communications among client, metadata and chunk servers are implemented by socket. Client sends request to metadata first, and then write or read files to multiple chunk servers based on the location information provided by metadata.

The simulation will be implemented in the Java programming language and compiled with Java compiler and run Linux or window platform.

## **Implementation**

The overall structure of our simulation is illustrated in the Figure 1. Three entities, Client, Metadata Server and Chunk Server, communicate with each other through Java socket connection. Client sends out command to either Metadata Server or Chunk Server and gets responds back from them later on. Metadata Server and Chunk Server can communicate with each other about node information. In the Metadata Server, there is a node map which contains all the node occupation status. All the file metadata are all also stored in it for file metadata operations. Chunk Server is mainly for file I/O. Each Node in it is simulated using a Java thread. Node threads are running parallely storing file chunks mimicking the real data center nodes.





**Figure 1. Overall simulation structure**

### I. Client.

Client is the controller interface where all the commands are entered. It sets up the socket server, parses the commands and communicates with Metadata Server and Chunk Server to get the job done. Commands are separated into two groups: those involve only Metadata Server (Table 1) and those involve both (Table 2).

<code>getDiskCapacity</code>	To get the total disk capacity in the distributed file system
<code>rename oldFileName newFileName</code>	To rename a file giving the old name and the new name
<code>getFileSize fileName</code>	To get the file size given the file name

<code>getFileStat fileName</code>	To get the file statistics including file name, file size and creation time given the file name
<code>getNodeStat</code>	To get the node load information including the total node disk size and the leftover disk size

**Table 1. Commands involve Metadata Server only**

<code>read -s/-p fileName</code>	To read a file given the file name. <code>-s</code> for sequential reading and <code>-p</code> for parallel reading. Request is first sent to Metadata Server to get the storage information about each chunk in the file. This information is in turn used by Client-Chunk Server communication to get the data chunks to complete the read.
<code>write -s/-p fileName</code>	To write a file given the file name. <code>-s</code> for sequential writing and <code>-p</code> for parallel writing. Request is first sent to Metadata Server to get the storage information about the original and copies of each chunk in the file. This information is in turn used by Client-Chunk Server communication to store the data chunks to complete the write.
<code>stopNode</code>	To stop the operation of a random node to mimic node failure in reality. Request is first sent to Metadata Server get the node number to be suspended. The node number is then

	sent to the Chunk Server to kill the corresponding thread.
--	--

**Table 2. Commands involve both Metadata Server and Chunk Server**

## II. Metadata Server

Metadata Server contains two list:

```
private static int[][] nodeMap;
private static HashMap<String, FileEntry> files;
```

nodeMap is a 2D integer array which contains all the node storage information. The row number is consistent with the node number of each Node in the Chunk Server. In our simulation, there are 16 Nodes, resulting in 16 rows in nodeMap. There are 2 columns in each row. The first column tells the current address to be written into. The second column tells the free space left on this node. The sum of the two columns is the total size of this node. nodeMap is used in “getDiskCapacity”, “getNodeStat”, and “write”.

A HashMap is used to store and search all the files in the system. A FileEntry class is used to hold the metadata of each file. The metadata includes the name, the size and the creation time of the file. FileEntry class also contains the addresses of all the chunks in the file, including both the original copy and the extra copies made to be used in case of node failure.

The detailed implementation of each request from the Client is described in Table 3.

getDiskCapacity	Calculate total disk capacity using nodeMap. Add up all the leftover size of each node and
-----------------	---

	return the sum.
<code>rename oldFileName newFileName</code>	Find the file from the files HashMap and call its <code>rename()</code> method to change the name.
<code>getFileSize fileName</code>	Find the file from the files HashMap and return the file size.
<code>getFileStat fileName</code>	Find the file from the files HashMap and return the file statistics.
<code>getNodeStat</code>	Return the total disk size and the leftover disk size of each node calculated from <code>nodeMap</code> .
<code>read -s/-p fileName</code>	Find the file from the files HashMap and return the location of the first available copy.
<code>write -s/-p fileName</code>	See if the file already exists. If yes, delete the old file first. Based on the <code>nodeMap</code> , distribute the file chunks onto each node and return the location information for all three copies.
<code>stopNode</code>	Randomly generate a node number between 0 and 15. Iterate through all the files and mark the invalid chunks caused by this node failure.

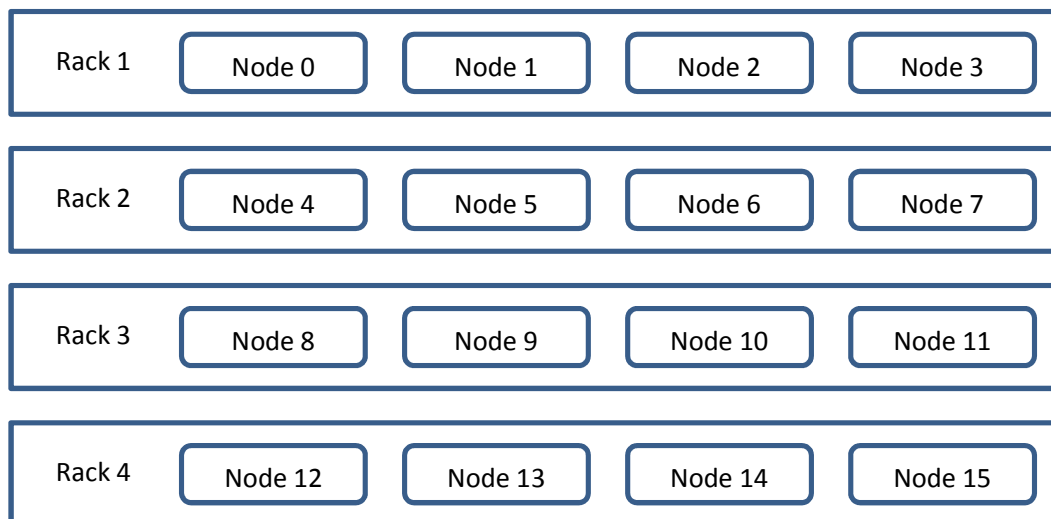
**Table 3. Command implementation in Metadata Server**

File writing handling in Metadata Server involves chunk distribution algorithm. In our system, file size is described in the number of integers in it. Files are chopped into chunks with 4 integer size and distributed over the Chunks Server nodes. Each chunk has three copies: one original copy, one on the next node and one on the node with the same position on the next rack. We assume that there are 4 racks in the Chunk Server center. Each rack contains 4 nodes to

compose a total of 16 nodes. For example, if node 1 is picked to store the original copy of a certain chunk, node 2 is used to store one of the copies to mimic the next node on the same rack and node 5 is used to store the other copy to mimic the node with the same position on the next rack. When distributing many chunks over the nodes, each node's free space is considered as its weight. With the known total disk capacity, the number of chunks stored on each node is calculated according to this weight ratio ( $length = totalNumberOfChunks * \frac{freeSpaceOnThisNode}{totalDiskCapacity}$ ). Thus a relatively even distribution is achieved. nodeMap is updated every time after a chunk is allocated.

### III. Chunk Server

Chunk Server created 16 threads to mimic 16 Nodes running at the same time in the server center. There are 4 racks and each rack contains 4 nodes. Each node has an ID, which is a number between 0 and 15. Node number  $4i \sim 4i+3$  are considered to be on the same rack (Figure 2). There are mainly three types of operations on Chunk Server: write, read, and stopNode.



**Figure 2. Node Placement Illustration**

Writing a file has two modes: sequential write mode and parallel write mode. The total elapsed time is computed and the two modes are compared with each other to learn how fast parallel writing is in distributed file system. If chunks need to be rewritten, the original chunks are deleted first. Otherwise, direct writing is done. Each chunk initializes a write thread, which talks to the node thread where the chunk should be written into. For sequential write, write threads are created and joined one at a time, whereas for parallel writing, all the write threads are created all together first and then joined all together later on. The code of sequential and parallel writing is shown below:

Sequential writing:

```
startTime = new Date();

String[] methods = opts[2].split("!");
data = opts[3].split(",");

String[] chunks, copies;

if (methods[0].equals("1"))
{
    chunks = methods[1].split(",");

    for (int i = 0; i < chunks.length; i++)
    {
        copies = chunks[i].split(",");

        for (int j = 0; j < copies.length; j++)
        {
            DeleteThread task = new DeleteThread(copies[j]);
            Thread thread = new Thread(task);
            thread.start();
            thread.join();
        }
    }
}

chunks = methods[2].split(",");

for (int i = 0; i < chunks.length; i++)
{
    copies = chunks[i].split(",");

    for (int j = 0; j < copies.length; j++)
```

```

    {
        WriteThread task = new WriteThread(i, copies[j]);
        Thread thread = new Thread(task);
        thread.start();
        thread.join();
    }
}

stopTime = new Date();

sendMessage("Elapsed Time: " + tidy.format((stopTime.getTime() - startTime.getTime()) / 1000.)
+ " seconds. Write complete!");

```

Parallel writing:

```

startTime = new Date();

ArrayList<Thread> threads = new ArrayList<Thread>();
String[] methods = opts[2].split("!");
data = opts[3].split(",");
String[] chunks, copies;

if (methods[0].equals("1"))
{
    chunks = methods[1].split(",");

    for (int i = 0; i < chunks.length; i++)
    {
        copies = chunks[i].split(",");

        for (int j = 0; j < copies.length; j++)
        {
            DeleteThread task = new DeleteThread(copies[j]);
            Thread thread = new Thread(task);
            threads.add(thread);
            thread.start();
        }
    }
}

chunks = methods[2].split(",");

for (int i = 0; i < chunks.length; i++)
{
    copies = chunks[i].split(",");

    for (int j = 0; j < copies.length; j++)
    {
        WriteThread task = new WriteThread(i, copies[j]);
        Thread thread = new Thread(task);
        threads.add(thread);
        thread.start();
    }
}

```

```

for (Thread thread : threads)
{
    thread.join();
}

stopTime = new Date();

sendMessage("Elapsed Time: " + tidy.format((stopTime.getTime() - startTime.getTime()) / 1000.)
    + " seconds. Write complete!");

```

Reading a file also has two modes: sequential read mode and parallel read mode. Very similar to writing a file, a read thread is created for each data chunk reading. The read thread communicates with node thread to do chunk reading. Sequential reading create and join one thread at a time and parallel reading creates all read thread and join all the them all together.

The code for reading a file is shown as below:

```

startTime = new Date();

String[] chunks = opts[2].split(";");
data = new String[chunks.length];
String result = "";
Thread[] threads = new Thread[chunks.length];

if (opts[1].equals("-s"))
{
    for (int i = 0; i < chunks.length; i++)
    {
        ReadThread task = new ReadThread(i, chunks[i]);
        threads[i] = new Thread(task);
        threads[i].start();
        threads[i].join();
    }
} else if (opts[1].equals("-p"))
{
    for (int i = 0; i < chunks.length; i++)
    {
        ReadThread task = new ReadThread(i, chunks[i]);
        threads[i] = new Thread(task);
        threads[i].start();
    }

    for (Thread thread : threads)
    {
        thread.join();
    }
} else
{

```



```
    sendMessage("Invalid command on " + serverName);  
    return;  
}  
  
for (int i = 0; i < data.length; i++)  
{  
    result += data[i];  
}  
  
stopTime = new Date();  
  
sendMessage("Elapsed Time: " + tidy.format((stopTime.getTime() - startTime.getTime()) / 1000.)  
    + " seconds. " + result.substring(0, result.length()-1));
```

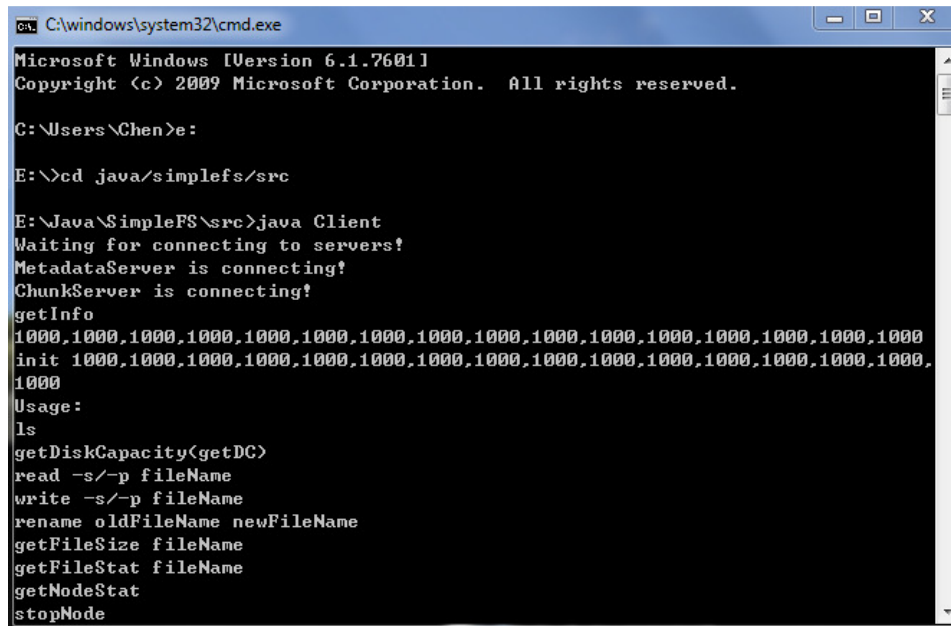
StopNode request is straight forward. The node with the designated ID is simply killed. No backup before its killing is needed because there are extra copies for all the chunks in the system. Single node failure should not affect file reading.

## Data analysis and discussion

### I. File System Initialization

Three terminals represent the client, Metadata Server and Chunk Servers. Below are the screenshots when our file system starts up.

- Client: Display a list of commands the file system supports and initial size of 16 nodes



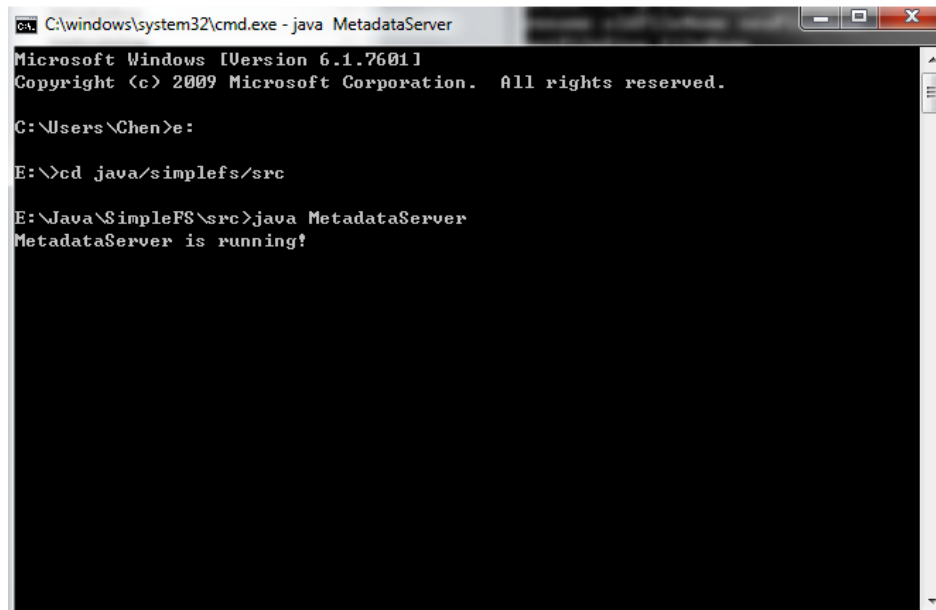
```
ca. C:\windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Chen>:

E:\>cd java/simplefs/src

E:\Java\SimpleFS\src>java Client
Waiting for connecting to servers!
MetadataServer is connecting!
ChunkServer is connecting!
getInfo
1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000
init 1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000
Usage:
ls
getDiskCapacity(getDC)
read -s/-p fileName
write -s/-p fileName
rename oldFileName newFileName
getFileSize fileName
getFileStat fileName
getNodeStat
stopNode
```

- Metadata Server: Running and waiting for requests from Client



```
ca. C:\windows\system32\cmd.exe - java MetadataServer
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Chen>:

E:\>cd java/simplefs/src

E:\Java\SimpleFS\src>java MetadataServer
MetadataServer is running!
```

- Chuck Server: There are 16 nodes running with the initial size of 1,000 each

```

C:\windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Chen>e:
E:\>cd java/simplefs/src
E:\Java\SimpleFS\src>java Server
ChunkServer is up!
Node 0 is running!
Node 2 is running!
Node 1 is running!
Node 3 is running!
Node 4 is running!
Node 5 is running!
Node 8 is running!
Node 7 is running!
Node 11 is running!
Node 6 is running!
Node 14 is running!
Node 13 is running!
A total of 16 nodes are running!
Node 10 is running!
Node 9 is running!
Node 12 is running!
Node 15 is running!
1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000

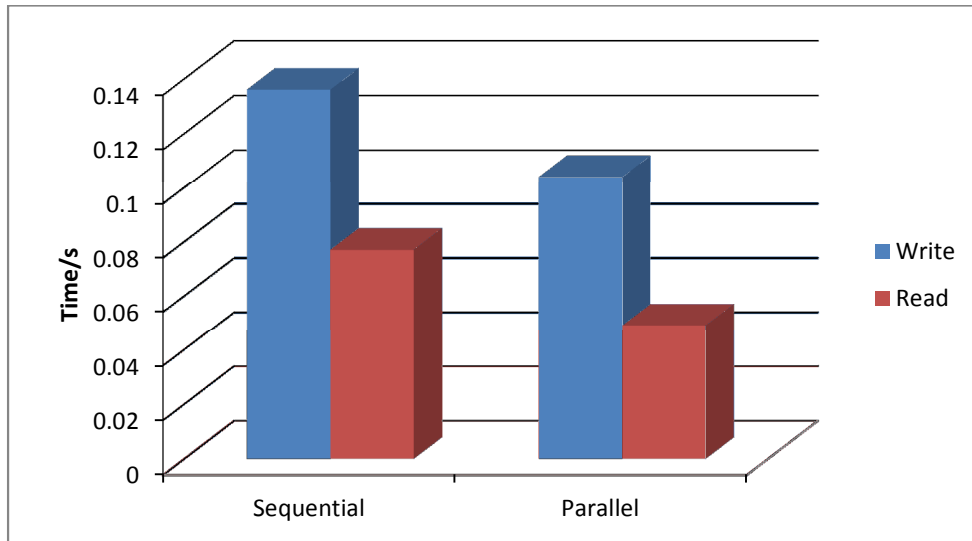
```

### Speed Analysis – Sequential vs. Parallel

As a distributed file system, file operations are running in parallel mode, which is implemented by 16 parallel threads in our chuck server. To compare the speed performance, we also add the sequential operation to write and read files.

The time taking for writing/reading the same file (file1.txt) with a size of 256 in sequential (write -s) and parallel (write -p) mode: average of 3 experiments on the same operation

Time/s	Sequential	Parallel	Time saving/%
Write	0.136	0.104	23.5
Read	0.077	0.049	36.4



### Conclusion

Generally, parallel writing/reading is much faster than sequential mode. In our system, sequential mode is implemented in one thread working sequentially, while parallel mode is implemented in 16 threads writing/reading parallel. However, the speed is not 16 times faster. It may be due to the overhead issue of managing multiple threads in Java. In addition, in the parallel mode, there is still one sequential element of parsing the storage chunk address from metadata service, which alpha is not 0 in the Amdahl's law.

Screenshot – write in sequential mode

```
write -s 1!0.49,1.19,4.19;0.50,1.20,4.20;0.51,1.21,4.21;1.22,2.19,5.18;1.23,2.20
,5.19;1.24,2.21,5.20;1.25,2.22,5.21;2.23,3.49,6.19;2.24,3.50,6.20;2.25,3.51,6.21
;2.26,3.52,6.22;3.53,4.22,7.18;3.54,4.23,7.19;3.55,4.24,7.20;4.25,5.22,8.18;4.26
,5.23,8.19;4.27,5.24,8.20;5.25,6.23,9.18;5.26,6.24,9.19;5.27,6.25,9.20;6.26,7.21
,10.18;6.27,7.22,10.19;6.28,7.23,10.20;7.24,8.21,11.18;7.25,8.22,11.19;7.26,8.23
,11.20;7.27,8.24,11.21;8.25,9.21,12.18;8.26,9.22,12.19;8.27,9.23,12.20;9.24,10.2
1,13.18;9.25,10.22,13.19;9.26,10.23,13.20;9.27,10.24,13.21;10.25,11.22,14.18;10.
26,11.23,14.19;10.27,11.24,14.20;11.25,12.21,15.48;11.26,12.22,15.49;11.27,12.23
,15.50;12.24,13.22,0.52;12.25,13.23,0.53;12.26,13.24,0.54;12.27,13.25,0.55;13.26
,14.21,1.26;13.27,14.22,1.27;13.28,14.23,1.28;14.24,15.51,2.27;14.25,15.52,2.28;
14.26,15.53,2.29;14.27,15.54,2.30;15.55,0.56,3.56;15.56,0.57,3.57;15.57,0.58,3.5
8;15.58,0.59,3.59;15.59,0.60,3.60;15.60,0.61,3.61;15.61,0.62,3.62;15.62,0.63,3.6
3;15.63,0.64,3.64;15.64,0.65,3.65;15.65,0.66,3.66;15.66,0.67,3.67;15.67,0.68,3.6
8;!0.69,1.29,4.28;0.70,1.30,4.29;0.71,1.31,4.30;1.32,2.31,5.28;1.33,2.32,5.29;1.
34,2.33,5.30;1.35,2.34,5.31;2.35,3.69,6.29;2.36,3.70,6.30;2.37,3.71,6.31;2.38,3.
72,6.32;3.73,4.31,7.28;3.74,4.32,7.29;3.75,4.33,7.30;4.34,5.32,8.28;4.35,5.33,8.
29;4.36,5.34,8.30;4.37,5.35,8.31;5.36,6.33,9.28;5.37,6.34,9.29;5.38,6.35,9.30;5.
39,6.36,9.31;6.37,7.31,10.28;6.38,7.32,10.29;6.39,7.33,10.30;7.34,8.32,11.28;7.3
5,8.33,11.29;7.36,8.34,11.30;7.37,8.35,11.31;8.36,9.32,12.28;8.37,9.33,12.29;8.3
8,9.34,12.30;8.39,9.35,12.31;9.36,10.31,13.29;9.37,10.32,13.30;9.38,10.33,13.31;
9.39,10.34,13.32;10.35,11.32,14.28;10.36,11.33,14.29;10.37,11.34,14.30;10.38,11.
35,14.31;11.36,12.32,15.68;11.37,12.33,15.69;11.38,12.34,15.70;11.39,12.35,15.71
;12.36,13.33,0.72;12.37,13.34,0.73;12.38,13.35,0.74;12.39,13.36,0.75;13.37,14.32
,1.36;13.38,14.33,1.37;13.39,14.34,1.38;14.35,15.72,2.39;14.36,15.73,2.40;14.37,
15.74,2.41;14.38,15.75,2.42;15.76,0.76,3.76;15.77,0.77,3.77;15.78,0.78,3.78;15.7
9,0.79,3.79;15.80,0.80,3.80;15.81,0.81,3.81;15.82,0.82,3.82;15.83,0.83,3.83; 1,2
,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8
,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,
0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3
,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9
,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,
5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4
,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54
Elapsed Time: 0.125 seconds. Write complete!
ChunkServer:
Elapsed Time: 0.125 seconds. Write complete!
```

Screenshot – write in parallel mode

```
write -p 1!0.25,1.10,4.10;0.26,1.11,4.11;0.27,1.12,4.12;1.13,2.9,5.9;1.14,2.10,5
.10;1.15,2.11,5.11;2.12,3.24,6.9;2.13,3.25,6.10;2.14,3.26,6.11;2.15,3.27,6.12;3.
28,4.13,7.9;3.29,4.14,7.10;3.30,4.15,7.11;4.16,5.12,8.9;4.17,5.13,8.10;4.18,5.14
,8.11;5.15,6.13,9.9;5.16,6.14,9.10;5.17,6.15,9.11;6.16,7.12,10.9;6.17,7.13,10.10
;6.18,7.14,10.11;7.15,8.12,11.9;7.16,8.13,11.10;7.17,8.14,11.11;8.15,9.12,12.9;8
.16,9.13,12.10;8.17,9.14,12.11;9.15,10.12,13.9;9.16,10.13,13.10;9.17,10.14,13.11
;10.15,11.12,14.9;10.16,11.13,14.10;10.17,11.14,14.11;11.15,12.12,15.24;11.16,12
.13,15.25;11.17,12.14,15.26;12.15,13.12,0.28;12.16,13.13,0.29;12.17,13.14,0.30;1
3.15,14.12,1.16;13.16,14.13,1.17;13.17,14.14,1.18;14.15,15.27,2.16;14.16,15.28,2
.17;14.17,15.29,2.18;15.30,0.31,3.31;15.31,0.32,3.32;15.32,0.33,3.33;15.33,0.34,
3.34;15.34,0.35,3.35;15.35,0.36,3.36;15.36,0.37,3.37;15.37,0.38,3.38;15.38,0.39,
3.39;15.39,0.40,3.40;15.40,0.41,3.41;15.41,0.42,3.42;15.42,0.43,3.43;15.43,0.44,
3.44;15.44,0.45,3.45;15.45,0.46,3.46;15.46,0.47,3.47;15.47,0.48,3.48;!0.49,1.19,
4.19;0.50,1.20,4.20;0.51,1.21,4.21;1.22,2.19,5.18;1.23,2.20,5.19;1.24,2.21,5.20;
1.25,2.22,5.21;2.23,3.49,6.19;2.24,3.50,6.20;2.25,3.51,6.21;2.26,3.52,6.22;3.53,
4.22,7.18;3.54,4.23,7.19;3.55,4.24,7.20;4.25,5.22,8.18;4.26,5.23,8.19;4.27,5.24,
8.20;5.25,6.23,9.18;5.26,6.24,9.19;5.27,6.25,9.20;6.26,7.21,10.18;6.27,7.22,10.1
9;6.28,7.23,10.20;7.24,8.21,11.18;7.25,8.22,11.19;7.26,8.23,11.20;7.27,8.24,11.2
1;8.25,9.21,12.18;8.26,9.22,12.19;8.27,9.23,12.20;9.24,10.21,13.18;9.25,10.22,13
.19;9.26,10.23,13.20;9.27,10.24,13.21;10.25,11.22,14.18;10.26,11.23,14.19;10.27,
11.24,14.20;11.25,12.21,15.48;11.26,12.22,15.49;11.27,12.23,15.50;12.24,13.22,0.
52;12.25,13.23,0.53;12.26,13.24,0.54;12.27,13.25,0.55;13.26,14.21,1.26;13.27,14.
22,1.27;13.28,14.23,1.28;14.24,15.51,2.27;14.25,15.52,2.28;14.26,15.53,2.29;14.2
7,15.54,2.30;15.55,0.56,3.56;15.56,0.57,3.57;15.57,0.58,3.58;15.58,0.59,3.59;15.
59,0.60,3.60;15.60,0.61,3.61;15.61,0.62,3.62;15.62,0.63,3.63;15.63,0.64,3.64;15.
64,0.65,3.65;15.65,0.66,3.66;15.66,0.67,3.67;15.67,0.68,3.68; 1,2,3,4,5,6,7,8,9,
9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5
,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,
5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,
67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,5
4,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,
6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67
,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54
Elapsed Time: 0.109 seconds. Write complete!
ChunkServer:
Elapsed Time: 0.109 seconds. Write complete!
```

Screenshot – read in sequential mode

```
read -s test.txt
read -s test.txt
0.69;0.70;0.71;1.32;1.33;1.34;1.35;2.35;2.36;2.37;2.38;3.73;3.74;3.75;4.34;4.35;
4.36;4.37;5.36;5.37;5.38;5.39;6.37;6.38;6.39;7.34;7.35;7.36;7.37;8.36;8.37;8.38;
8.39;9.36;9.37;9.38;9.39;10.35;10.36;10.37;10.38;11.36;11.37;11.38;11.39;12.36;1
2.37;12.38;12.39;13.37;13.38;13.39;14.35;14.36;14.37;14.38;15.76;15.77;15.78;15.
79;15.80;15.81;15.82;15.83;
read -s 0.69;0.70;0.71;1.32;1.33;1.34;1.35;2.35;2.36;2.37;2.38;3.73;3.74;3.75;4.
34;4.35;4.36;4.37;5.36;5.37;5.38;5.39;6.37;6.38;6.39;7.34;7.35;7.36;7.37;8.36;8.
37;8.38;8.39;9.36;9.37;9.38;9.39;10.35;10.36;10.37;10.38;11.36;11.37;11.38;11.39
;12.36;12.37;12.38;12.39;13.37;13.38;13.39;14.35;14.36;14.37;14.38;15.76;15.77;1
5.78;15.79;15.80;15.81;15.82;15.83;
Elapsed Time: 0.078 seconds. 1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9
```

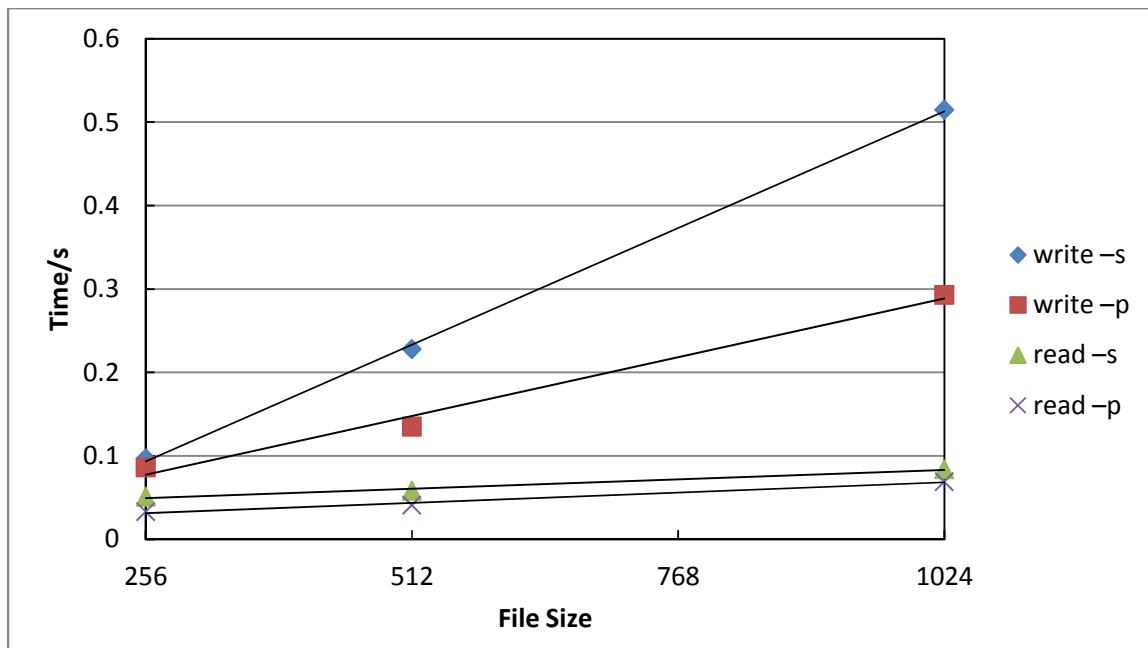
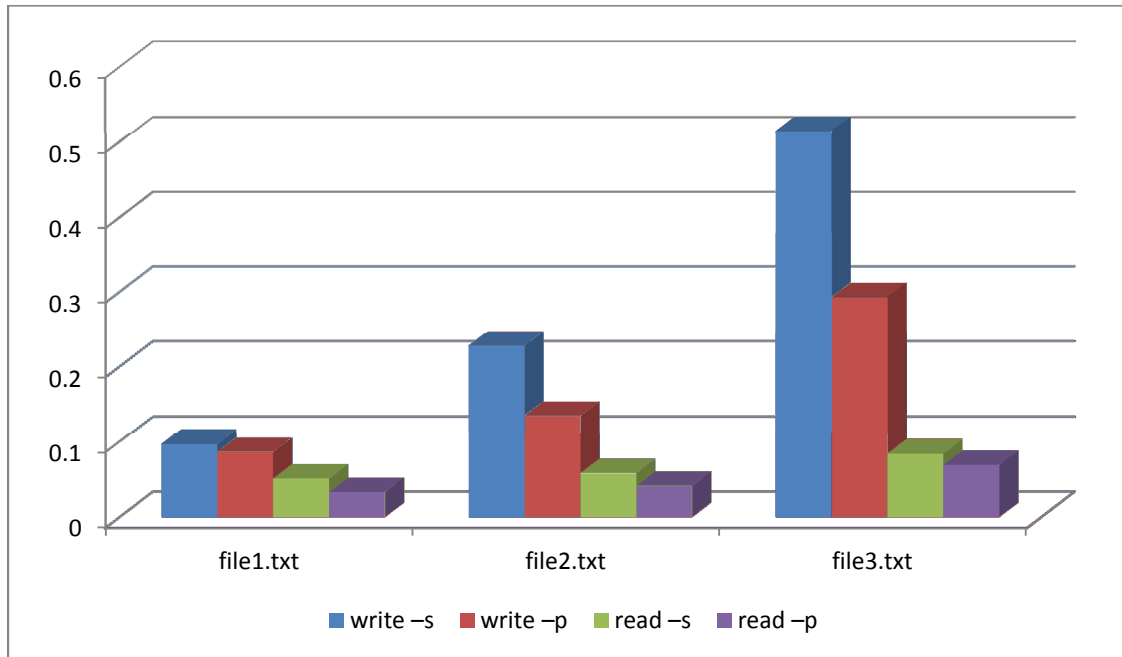
Screenshot – read in parallel mode

```
read -p test.txt
read -p test.txt
0.69;0.70;0.71;1.32;1.33;1.34;1.35;2.35;2.36;2.37;2.38;3.73;3.74;3.75;4.34;4.35;
4.36;4.37;5.36;5.37;5.38;5.39;6.37;6.38;6.39;7.34;7.35;7.36;7.37;8.36;8.37;8.38;
8.39;9.36;9.37;9.38;9.39;10.35;10.36;10.37;10.38;11.36;11.37;11.38;11.39;12.36;1
2.37;12.38;12.39;13.37;13.38;13.39;14.35;14.36;14.37;14.38;15.76;15.77;15.78;15.
79;15.80;15.81;15.82;15.83;
read -p 0.69;0.70;0.71;1.32;1.33;1.34;1.35;2.35;2.36;2.37;2.38;3.73;3.74;3.75;4.
34;4.35;4.36;4.37;5.36;5.37;5.38;5.39;6.37;6.38;6.39;7.34;7.35;7.36;7.37;8.36;8.
37;8.38;8.39;9.36;9.37;9.38;9.39;10.35;10.36;10.37;10.38;11.36;11.37;11.38;11.39
;12.36;12.37;12.38;12.39;13.37;13.38;13.39;14.35;14.36;14.37;14.38;15.76;15.77;1
5.78;15.79;15.80;15.81;15.82;15.83;
Elapsed Time: 0.047 seconds. 1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9
```

## II. Speed Analysis – Different File Size

In this section, we conduct writing and reading in both sequential and parallel mode on 3 files with different size, and below is the result table of time comparison:

File Name	Size	write -s (sec)	write -p (sec)	read -s (sec)	read -p (sec)
file1.txt	256	0.097	0.086	0.051	0.033
file2.txt	512	0.228	0.135	0.058	0.041
file3.txt	1,024	0.515	0.293	0.084	0.069



### Conclusion

The comparisons result in generally in line with our expectation.

- Reading is faster than writing, as system makes duplicate copies during the writing process
- For the same file, parallel mode is faster than sequential mode



- For the same operation, with the increase of file size, the time spent also increases accordingly
- For the operations which requires longer period of time, the relationship between time spent and file size is approximately linear, as overheads play a smaller roles in the comparison

### III. Load Balance Analysis

- Initialization status: 16 nodes are initialized with the same size at 1,000

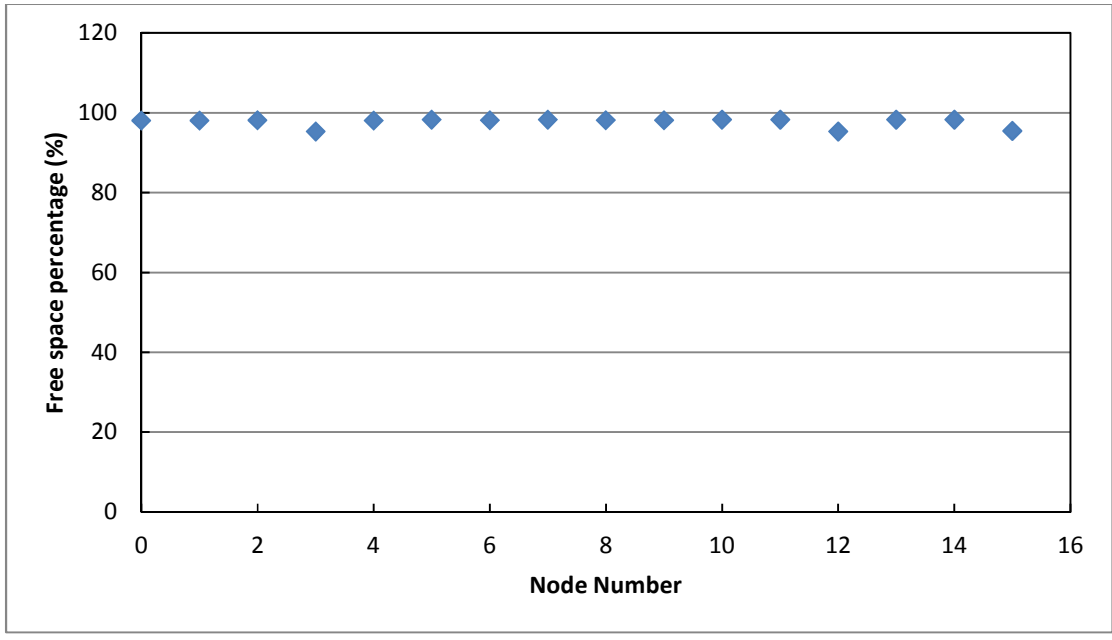
```

getDiskCapacity
getDiskCapacity
Client: getDiskCapacity
16000
MetadataServer:
16000
getNodeStat
getNodeStat
1000.1000;1000.1000;1000.1000;1000.1000;1000.1000;1000.1000;1000.1000;1000.1000;
1000.1000;1000.1000;1000.1000;1000.1000;1000.1000;1000.1000;1000.1000;
Node    Total    Leftover
0       1000    1000
1       1000    1000
2       1000    1000
3       1000    1000
4       1000    1000
5       1000    1000
6       1000    1000
7       1000    1000
8       1000    1000
9       1000    1000
10      1000    1000
11      1000    1000
12      1000    1000
13      1000    1000
14      1000    1000
15      1000    1000

```

- Load Balance: After the operation of writing test.txt file twice (once in sequential and once in parallel), below is the updated disk capacity and node status:

```
C:\windows\system32\cmd.exe - java Client
getDiskCapacity
getDiskCapacity
Client: getDiskCapacity
15616
MetadataServer:
15616
getNodeStat
getNodeStat
1000.980;1000.980;1000.981;1000.953;1000.980;1000.982;1000.981;1000.982;1000.981
;1000.981;1000.982;1000.982;1000.953;1000.982;1000.982;1000.954;
Node    Total    Leftover
0       1000    980
1       1000    980
2       1000    981
3       1000    953
4       1000    980
5       1000    982
6       1000    981
7       1000    982
8       1000    981
9       1000    981
10      1000    982
11      1000    982
12      1000    953
13      1000    982
14      1000    982
15      1000    954
```



### Conclusion

As shown in the result, after writing a file, the total disk capacity dropped from 16,000 to 15,616. Meanwhile, the load of each node remained pretty balanced. Most of nodes are at 98% capacity and others are at 95% capacity. The difference may be due to some nodes keeping the additional duplicate copies of file chunks. The algorithms we chose in the metadata service to allocate the storage address can achieve the load balance requirement for a simulated distributed file system.

## **IV. Fault Tolerance Analysis**

To simulate the random node failure, we implemented “stopNode” command in our system. After client types in this command, the system will randomly kill a running thread in the chunk server, which simulates a node failure. Then client can try to use read command to get the previously stored file to test whether the file can still be retrieved or not.

### Conclusion

As shown in the result screen below, our distributed file system can support file operation with random node failure. In this case, node 15 was stopped working, and client can still read the test.txt file from the file system successfully.

Actually, in our simulation, we can run “stopNode” command multiple times, which simulates multiple nodes failure at this same time, and client can still retrieve the file, so our simulated distributed file system achieves the fault tolerance by storing duplicates of file record on different nodes.

```
C:\windows\system32\cmd.exe - java Client
ls
Client: ls
test.txt
MetadataServer:
test.txt
stopNode
stopNode
15
stopNode 15
Client: stopNode
Node 15 has stopped working!
ChunkServer:
Node 15 has stopped working!
read -p test.txt
read -p test.txt
0.49;0.50;0.51;1.22;1.23;1.24;1.25;2.23;2.24;2.25;2.26;3.53;3.54;3.55;4.25;4.26;
4.27;5.25;5.26;5.27;6.26;6.27;6.28;7.24;7.25;7.26;7.27;8.25;8.26;8.27;9.24;9.25;
9.26;9.27;10.25;10.26;10.27;11.25;11.26;11.27;12.24;12.25;12.26;12.27;13.26;13.2
7;13.28;14.24;14.25;14.26;14.27;0.56;0.57;0.58;0.59;0.60;0.61;0.62;0.63;0.64;0.6
5;0.66;0.67;0.68;
read -p 0.49;0.50;0.51;1.22;1.23;1.24;1.25;2.23;2.24;2.25;2.26;3.53;3.54;3.55;4.
25;4.26;4.27;5.25;5.26;5.27;6.26;6.27;6.28;7.24;7.25;7.26;7.27;8.25;8.26;8.27;9.
24;9.25;9.26;9.27;10.25;10.26;10.27;11.25;11.26;11.27;12.24;12.25;12.26;12.27;13
.26;13.27;13.28;14.24;14.25;14.26;14.27;0.56;0.57;0.58;0.59;0.60;0.61;0.62;0.63;
0.64;0.65;0.66;0.67;0.68;
Elapsed Time: 0.046 seconds. 1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9
,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,
5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4
,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9
,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,
54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5
,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,6
7,8,9,0,5,54
ChunkServer:
Elapsed Time: 0.046 seconds. 1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9
,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,
5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4
,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9
,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,
54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5
,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,6
7,8,9,0,5,54
```

## Conclusions and recommendations

### I. Summary and Conclusions

As discussed in the previous data analysis section. Our simulated distributed file system supports parallel write / read files, maintain the load balance across 16 nodes and achieves basic fault tolerance on multiple random node failures.

In addition, our system also support other basic operations as a file system, including rename a file, get the file size, etc. Below is a full list of commands we support:

```
Usage:
ls
getDiskCapacity
read -s/-p fileName
write -s/-p fileName
rename oldFileName newFileName
getFileSize fileName
getFileStat fileName
getNodeStat
stopNode
```

## II. Future studies

There are also several areas we can improve on our simulation. For future works, we could develop our system to support multiple clients operating on the same time. To make the system more efficient, we could also implement a garbage collector function to release the waste space periodically.

For further data analysis purpose, we could change the algorithm of the metadata service to allocate storage space across nodes in several different ways, and then to compare the efficiency of different algorithms regarding the load balance. With this established simulated system, we can conduct further research on various topics in the future.

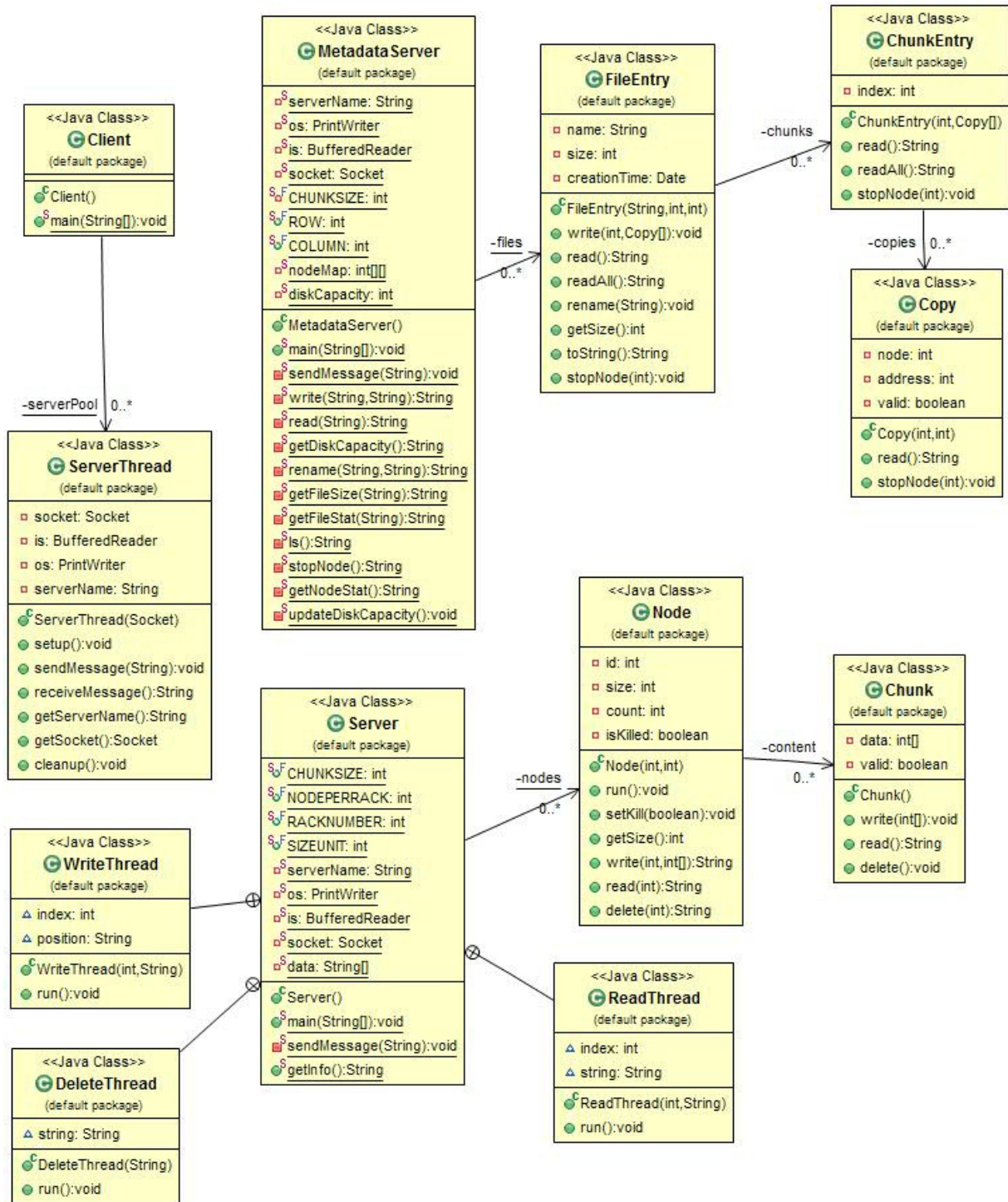
## Bibliography

[1]. Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.

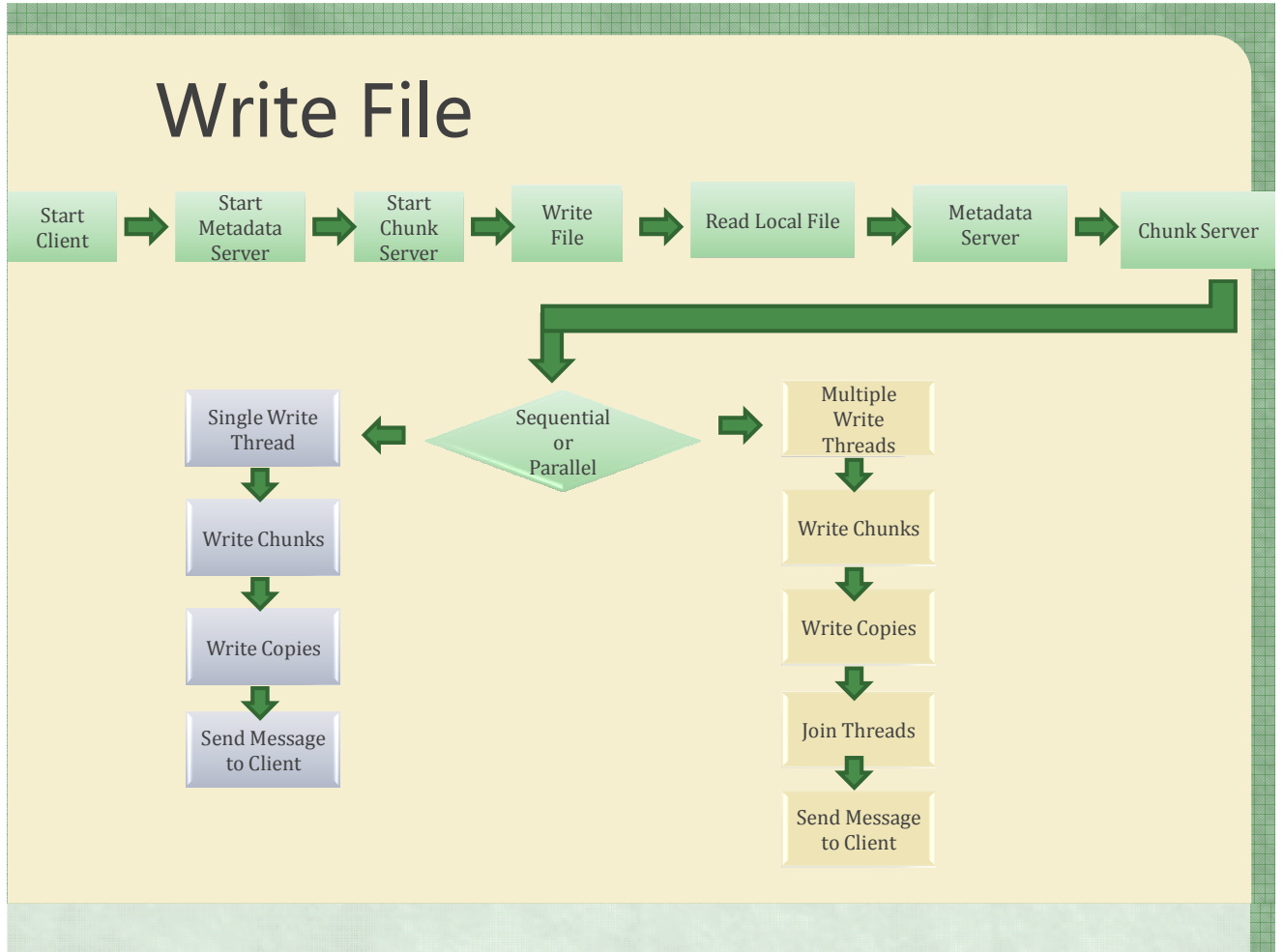
- [2]. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. SOSP'03, October 19–22, 2003
- [3]. D. Fetterly, M. Haridasan, M. Isard, and S. Sundararaman, "Tidyfs: a simple and small distributed file system," in Proceedings of the 2011 USENIX conference on USENIX annual technical conference, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 34–34.
- [4]. Chien-Ming Wang, Chi-Chang Huang, Huan-Ming Lian. ASDF: An Autonomous and Scalable Distributed File System Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on. 23-26 May 2011
- [5]. Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In Proceedings of the 7<sup>th</sup> Symposium on Operating Systems Design and Implementation (OSDI). 2006. USENIX.
- [6]. DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI) (San Francisco, CA, USA, 2004).
- [7]. File appends in HDFS. <http://www.cloudera.com/blog/2009/07/file-appends-in-hdfs/>.
- [8]. GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In Proceedings of the 19th Symposium on Operating Systems Principles (SOSP) (Bolton Landing, NY, USA, 2003).
- [9]. Hadoop wiki. <http://wiki.apache.org/hadoop/>, April 2008.
- [10]. ISARD, M., BUDI, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In Proceedings of 2nd European Conference on Computer Systems (EuroSys) (Lisbon, Portugal, 2007).
- [11]. BORTHAKUR, D. HDFS architecture. Tech. rep., Apache Software Foundation, 2008.
- [12]. SHVACHKO, K. V. HDFS scalability: The limits to growth. ;login 35, 2 (April 2010).

# Appendices

Class Diagram:

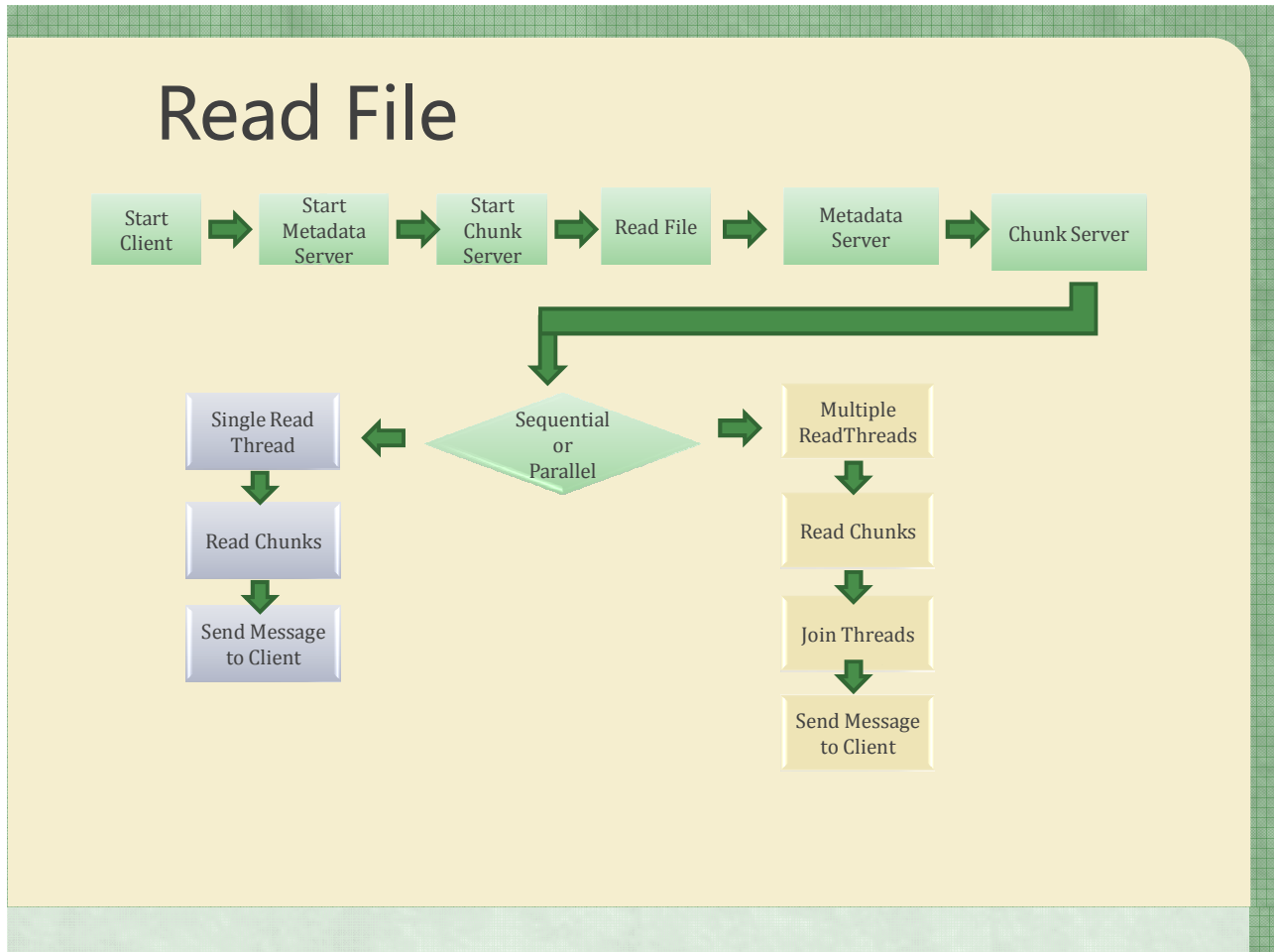


## Simulation Work Flow for Write File





## Simulation Work Flow for Read File



Test File1:

1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,5  
4,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5  
,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0  
,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9  
,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8  
,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54



6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,  
5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,  
4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,  
3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,  
2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,  
1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,5  
4,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5  
,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0  
,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9  
,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54,1,2,3,4,5,6,7,8,9,9,67,8,9,0,5,54