A Distributed File System for Frequency Reading of Various File Sizes

{Pengfei MA, Yanshen Yin, Chao Lan}
Department of Computer Science and Technology
Tsinghua University
Beijing, China
{mpf07, yys12, lanc11}@mails.tsinghua.edu.cn

{Yong Zhang, Chunxiao Xing}
Research Institute of Information Technology
Tsinghua University
Beijing, China
{zhangyong05, xingcx}@tsinghua.edu.cn

***Abstract*—The Web File Management System (WFMS) is a distributed file system, which is designed for frequency reading of various file sizes. It consists of one single master and multi data servers. We use blocks to store data, which contains different size of chunks. The block/chunk mechanism achieves nearly 100% utilization of block. We also design hot block to raise the reading speed of frequency blocks through increasing the replicas automatically. WFMS works on the web exposing RESTful services and providing authority control. Besides, we implement our search engine with Lucene which provides query service by file name, file size, file type, date, owner and so on. Until now, our system has stored over 40TB data which contains more than 2 million files and it has worked stably for two years.**

Keywords- WFMS; distributed file system; file system

I. INTRODUCTION

The Web File Management System (WFMS) is a distributed file system designed for storage and management of massive data. It is designed to provide efficient, reliable, convenient, and safe multi-user storage service, as well as provides platform for further study on big data. WFMS supports the following operations: write, read, delete and query.

WFMS consists of a single master and multiple data servers. There're two types of data: file system metadata which is stored in the master, application data which is stored in the data servers. For better management, we use blocks to store data, which contains chunks of different sizes. The block/chunk mechanism achieves nearly 100% utilization of block. We also design hot block to boost the reading speed of frequency blocks through increasing the number of replicas automatically. Block schedule and file access control is critical in our system which will be discussed later. WFMS works on the web exposing RESTful services, which support file system operations. It offers two access models: accessing via web using browser and accessing via WFMS client which can mount WFMS to a local drive using Dokan Library. The system also provides multi-level authorities, such as administrator and normal user. Besides, we implement our search engine with Lucene. Users can search contents by file name, file size, file type, date, owner etc. Until now, our system has stored over 40TB data which contains more than 2 million files and it has worked stably for two years.

The rest of the paper is organized as follows. In section 2, we show some related work about distributed file system. In section 3, we present the design of our system in detail. In section 4, we describe the system interactions about write and replica. In section 5, we show the experiment results on real system. Finally, we conclude this paper in section 6.

II. RELATED WORK

Many existing distributed file systems have been raised. The reason for us to design a new distributed system rather than utilizing an existing system is that the existing distributed file system such as GFS[1], HDFS[2], PVFS[3], Lustre is not satisfied with our requirements. WFMS should not only support the basic features of distributed file system, but also provide some new characteristics according to the application requirements.

In our system, we provide our services on RESTful for all the operations, such as read, write, delete, query etc. It's more convenient and compatible for 3rd application to invoke than invoking through bare sockets. WFMS mainly focus on two points: (1) we want to put forward a good storage mechanism which can well support both big files and small files, with high space utilizations; (2) we want to come up with a speedup-strategy for reading of frequency read files.

Google designed and implemented the Google File System (GFS)[4] to meet the rapidly growing demands of applications in company. GFS consists of a single master and multiple chunk servers. It can automatically handle kinds of component failures, support appending modifications to files rather than overwriting. However, GFS is unwieldy to manage lots of approximately KB-sized files, which couldn't be tolerated in our applications.

HDFS is the file system component of Hadoop[5]. It is specially designed for applications in MapReduce framework. As in GFS, HDFS stores metadata on NameNode and application data on DataNode. It provides data replica for data reliability and image/journal mechanism for data recovery, and has gained a very big success all over the world.

PVFS is short for Parallel Virtual File System, which is designed for high-bandwidth concurrent writes that parallel complicated applications typically require. It is not unsuitable for distributed access to files from multiple client machines. PVFS is also can't satisfy our requirements.

978-1-4799-3219-1/13 $31.00 © 2013 IEEE
DOI 10.1109/WISA.2013.70

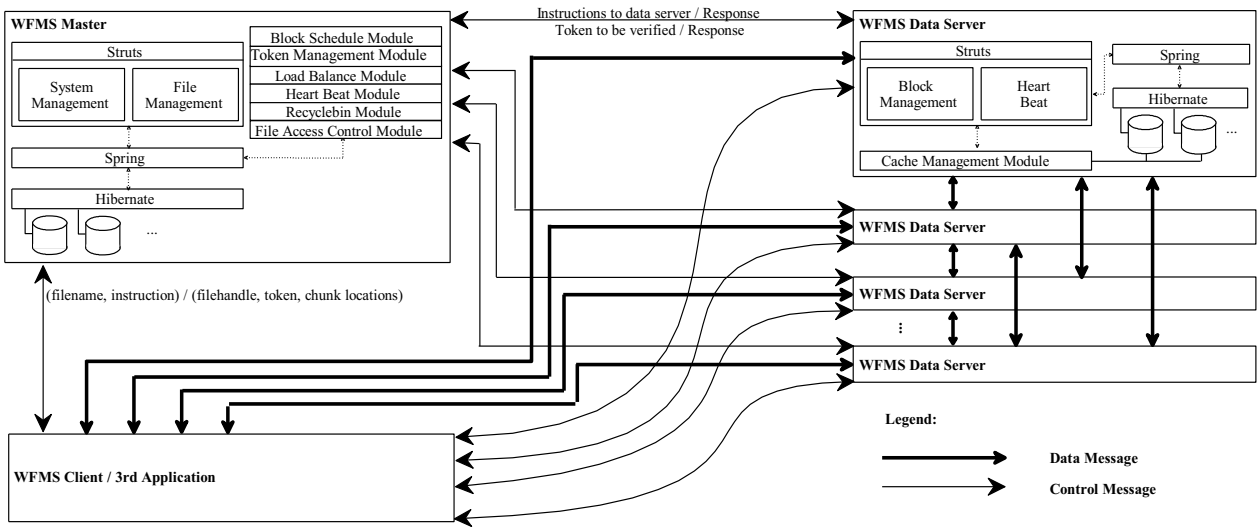

Figure 1. WFMS Architecture

Besides, lots of other distributed file systems were developed such as Dynamo file system, Mogile file system, Taobao file system, Tecent file system etc.

WFMS's architecture is similar to GFS and HDFS, single master and multiple data servers, and data is stored in blocks. We also did some further design to let WFMS can handle lots of small files and achieve better read and write performance.

III. DESIGN

Web File Manage System (WFMS) is a REST-based distributed file system developed in Java. It is designed to provide efficient, reliable, convenient and safe storage service using large clusters of commodity hardware.

WFMS works on the web exposing RESTful services, which support file system operations. It offers two access modes: accessing via web using browser and accessing via WFMS client which can mount WFMS to local drive using Dokan library.

A. Architecture

A WFMS cluster consists of a single master and multiple data servers, as shown in Figure 1. Each of these is typically a commodity Linux machine running a web service process under Tomcat.

A WFMS cluster consists of multiple nodes. These nodes are divided into two types: one Master and a large number of Data Servers. Each file is split into fixed-maximum-size chunks, and these chunks are contained in fixed-size blocks, which is stored in data server. Each block is replicated on several data servers throughout the network, with the default replicate times being two, and the default fixed-size being 64 megabytes. A block with high read frequency is called a hot block. WFMS will dynamically increase the copies of the hot blocks to improve the read efficiency. Vice versa, we call the block with low read frequency as cold block, and cold blocks have fewer copies to save more storage space.

The master doesn't store the actual blocks, but rather all the metadata, such as the tables mapping the file to chunk and block locations, the locations of the copies of the blocks, the attributes (ID, size, name, path, owner, create/access/modify time, status) of files, the information (ID, IP, port, public/private key, total/free space, status) of data servers, the information of user and role authorities, the index of file attributes (we use Lucene as the search engine) and so on. All the metadata is kept by the master and periodically updated.

WFMS exposes RESTful services as the API, and provides a website and a client (WFMS client) for users to access. Using the website, user can configure system (data server configuration, environment variable, and monitor configuration), upload/ download/ search/ undelete files. Using the WFMS client, user can access the file system as a local drive.

B. Master

A single master vastly simplifies our design and enables the master to make complicated block schedules and replication decisions. However, we must minimize its reads and writes so that it does not become a bottleneck. In WFMS, client never read and write file data through the master. Instead, a client gets token and chunk locations of a file from the master, then read and write on data servers directly using the token. When a read or write request is asked at the first time, the data server will ask master to verify the token and cache the verify result for a limited time.

We implement master in SSH framework, because of its lightweight and good performance. In the framework, some threads run as the separate modules to schedule the blocks, mange the caches and tokens, balance the load, control the file accesses, etc. The master will send a heart-beat request to all data servers every one minute to collect the real time status of each data server, including the usage of disk space, the health of the blocks and disks, the load of the system, etc.

The metadata is stored in MySQL. It's easier for us to manage the metadata (both insert and query) and ensure data consistency. Most importantly, MySQL is not likely to be a read or write bottleneck in our experiments.

C. Data Server

Data servers are implemented in SSH framework, and the main work of it is how to read or write a block more efficiently according to bid (block id), offset and length. Each block has a global unique id (bid), and corresponds to a fixed-sized (default-fixed-size = 64 MB) physical file on the disk. The global unique does not mean there is no more than one block with a specified bid in all data servers. Instead, you may find many blocks in different data servers with the same bid, because of the system redundancy.

Chunk is more fine-grained than block, and it's a variable sized section of block. Considering a file with less than 64MB, the file may be stored in one block, but obviously will not take up the wholly block. In this case, we call the section that the file occupies in the block as a chunk. So, a block may contain many files, while a chunk just contains the data of only one file as Figure 2 shows.

The block id is a 32 bytes UUID, which is separated by each 2 bytes to create a directory where the block stored. Besides, the snapshot and the file lock of the block are contained in the same directory. We use FileChannel (A Java NIO Class) to read and write blocks, which can map a region of a file into memory directly, and this is often much more efficient than invoking the usual read or write methods. File Channels are safe for use by multiple concurrent threads, and can read and write different regions of a file at the same time.

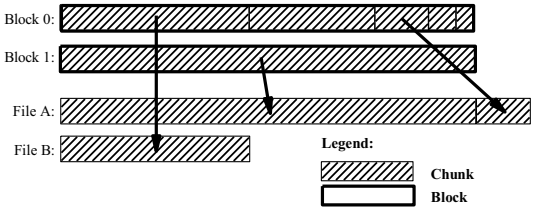

Figure 2. Block and Chunk

D. Client

Client invokes 17 RESTful services shown in Table 1 which are provided by master and data server. Secret key plays an important role in the interactions between client and master. At the beginning of service requests, a client needs to invoke Login service to get a temporary access key which we called secret key. The secret keys are managed by the master. Then, the client can get services from master using the secret key to verify user identity. A client need to offer master a heart-beat constantly to keep the secret key valid, because a secret key only has 5 minutes' live time without heart-beat, and the current loads of data servers will be returned as the response.

Before invoking the Read, Write or SetEOF services, a CreateFile service should be invoked first with arguments of file mode, file access, file share, file name, etc. CreateFile is very like the "fopen" function in C++, and you can get the file access token and chunk locations as the response. Then, the client can communicate with data servers to read, write or set end of file using the token. Finally, after the operations, the client should invoke CloseFile service to close the file. The invoking relations are shown in Figure 3.

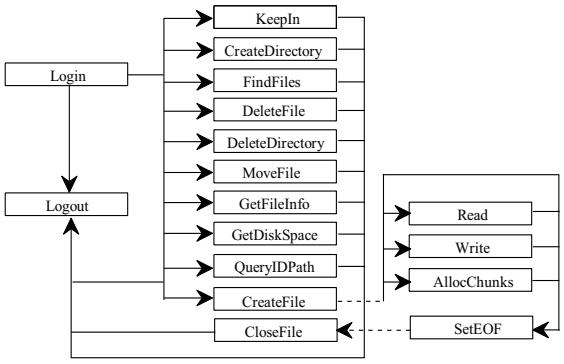

Figure 3. The Invoking Relations among The Interfaces

TABLE I. RESTFUL SERVICE INTERFACES

Provider	*RESTful Service*	*Request*	*Response*
Master	Login	username, md5(password)	secret key
	Logout	SecretKey	Result
	KeepIn	SecretKey	balanced data server list
	CreateFile	secretKey, filename, mode	token, chunks
	CloseFile	secretKey, token	result
	CreateDirectory	secretKey, filename	result
	DeleteFile	secretKey, filename	result
	DeleteDirectory	secretKey, filename	result
	FindFiles	secretKey, filename	file list
	GetDiskSpace	SecretKey	system disk space
	GetFileInfo	secretKey, filename	file attribute
	MoveFile	secretKey, filename, newname	result
	AllocChunks	secretKey, token, length	chunks
	SetEOF	Token, length	result
	QueryIDPath	secretKey, key, value	reuslt
Data Server	Read	Token, Bid, offset, length	data
	Write	Token, bid, offset, length,data	result

Let us explain the interactions for a simple read with reference to Figure 1. First, the client should send a Login-request and KeepIn-request to get a valid secret key and the load balanced data server list. Then, it sends the master a CreateFile-request containing the file name, mode and secret key. The master replies with the corresponding file access token and chunk locations. The Client then sends Read-requests to the data servers where the chunks located. The

requests specify the token, chunk location, and a byte range to read. Further reads of the same file require no more client-master interaction until the file is reopened. In fact, a same request will be also send to other data servers who have the replica. When reading finished, the client sends master a CloseFile-request to close the file.

E. Metadata

The master stores eight major types of metadata: the file information, the block information, the chunk information, data server information, system management information, log information, user information and lucence index information. A file information record contains the file's attribute, status and the chunk ids (cid) where the file stored. A block information record contains the block size, status and the data server ids where the block and its copies stored. A chunk information record contains the bid the chunk belongs to and the region of the chunk in the block. A data server information record contains the properties of one data server, such as IP, port, private/public key, total/used space, status, etc. System management information contains monitor details or system environment variables. Log information contains the logs of system operations to ensure the system consistency. User information contains the users' names, passwords and authorities. Lucene index information contains the indexes of files for searching.

In order to simplify the system realization, we persist the above seven major types (except lucence index information, it is persisted in its own file format) of metadata in MySQL. It's easier for us to manage the metadata (both insert and query) and ensure data consistency. Most importantly, MySQL is not likely to be a read or write bottleneck in our experiments. In fact, we will use memory caches for metadata to improving the efficiencies.

Besides the eight major types of metadata above, there are two non-persisted types of metadata: token information and file access control information. These two types of metadata are kept in memory and will not be persisted into disk. Token information is managed in a LRU cache, and it contains the tokens for the client to operate chunks on data servers. When a data server sends the master a request to verify a token, the token will be hit once if exist. And when a client sends the master a request to close file, the corresponding token will be set invalid. File access control information is managed in a memory cache by File Access Control Module, which is in charge of dealing with read-write conflict.

F. File Access Control

CreateFile service is a previous invoking before reads and writes, and CloseFile service is a later invoking after reads and writes. A CreateFile request requires arguments of file mode, file access, file share, file name, secret key, etc. File mode determines how to open or create the file, and it's a member of {CreateNew, Create, Open, OpenOrCreate, Truncate, Append}. File access determines how the file can be accessed, and it's a member of {Read, Write, ReadWrite}. File share determines how the file will be shared by processes, and it's a member of {None, Read, Write, ReadWrite, Delete, Inheritable}. Hence, the master knows who are reading/writing the files and the ways the files opened.

File Access Control Module manages the file access control information in a memory cache. It will update the information when a CreateFile or CloseFile request is called, and delete the timeout records constantly. The structure of file access control information is shown in Figure 4.

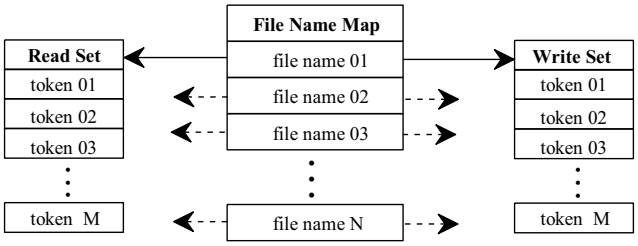

Figure 4. Structure of File Access Control Information in Memory

G. Block Schedule

Block schedule module has four responsibilities: allocating new blocks on data servers, assigning the blocks to write files, recycling the useless block to release space, and the block defragmentation. The system will load all available block metadata into a 2-level-map structure in memory when the system startup, which is shown in Figure 5. The first level contains the available blocks, which ordered by the maximal chunk-size of itself, and each block contains a block-read-write lock. The second level contains all available chunks of one block, which ordered by the chunk-size, and each chunk contains a synchronized lock. Our System will ensure the size of all available blocks around a specified value (default value is 10 GB), so we don't have to worry about there are too many blocks to be loaded in the memory.

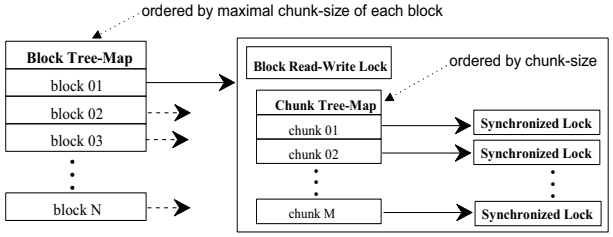

Figure 5. Structure of Block Schedule Module in Memory

While a write request coming, the module will assign some chunks to the file according a specified size. The part larger than the block size will assign to a single block, and the rest part will assign to a chunk which is a bit larger the data, or several small chunks. These assigned chunks will be deleted from the map, and put into chunk lease pool in memory. The chunk lease pool contains the chunks which have been assigned, and the chunks which haven't been used when the lease time out will be put back into block schedule map.

The delete operations will generate a large number of fragments, and it will lead to a block contains several discontinuous chunks. Especially, the chunks can be very

small when the deleted file is small. In our system, we will merge the small chunks together in each block.

IV. SYSTEM INTERACTIONS

A. Writing

When a file opened with OnlyRead mode, the chunks where the file located will be returned through CreateFile request, and then it's easy to read the file. However, a write operation may extend the file length, so additional chunks need to be allocated to the file. In the system, we can call AllocChunks-request to append extra chunks to a file, and call SetEOF-request to set file length.

In Figure 6, we illustrate the process through these numbered steps.

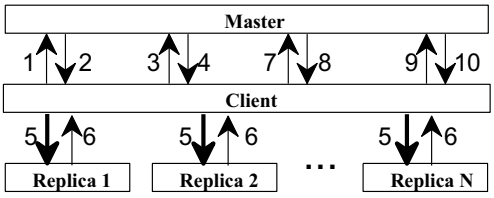

Figure 6. Write Control

1. CreateFile: the client asks the master which data servers store the chunks of the file.
2. The master replies the locations of each chunk.
3. AllocChunks (optional): the client asks the master allocate more chunks for the file. This it optional, because not all writing operation can extend the file length. The client can call this service several times when the space is not enough to write, and in order to release the call times, client can allocate much more chunks in one request.
4. The master replies the allocated chunks.
5. Write: The client pushes the data to all the replicas.
6. The data server replies to client that it has completed the operation.
7. SetEOF: When all operation completed, the client asks the master to set the file length and release the left allocated chunks.
8. The master replies the operation result.
9. CloseFile: the client asks the master to close the file, and release the file handle.
10. The master gives the operation result.

B. Replica

In order to ensure the reliability and improve the read performance, the block replicas are extremely necessary. In WFMS, each block is replicated on several data servers throughout the network, with the default replicate times being two, and the default fixed-size being 64 megabytes. The number of the replicas of each block can adjust dynamically. As the read frequency of a block increases, the number of the block copies will be increased dynamically. The block with high read frequency we call it hot block, and more copies help to improve the read efficiency. Vice versa, we call the block with little read frequency as cold block, and fewer copies help to save more storage space.

While a block is read in a high frequency at a data server, the data server will consider the block as a hot block, and asks the master to create a new copy of the block. Then the master checks whether the block is being written, if not the block will be locked with ReadOnly lock for a moment, and command the data server to do the replication. Finally, when replication finished, the master will unlock the block.

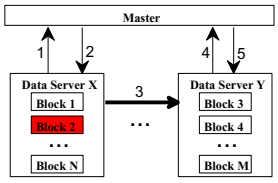

Figure 7. Write Control

In Figure 7, we illustrate the process through the following steps.

1. Data server X finds that the block 2 has been read frequently in recent time, then it consider the block to be hot, and send a request to master to do replication.
2. When the master received the request, it writes a replication-start log and judges whether it's necessary to do the replication in global. If necessary, then it gives the block a ReadOnly lock in access control module, and send the replication information back to data server X.
3. Data server X send the block 2 to data server Y
4. Data server Y notifies the master that he has received the whole block.
5. When master get the message, it writes a replication-end log, unlocks the block and gives a response to data server Y. Especially, if one replication hasn't been finished for a long time, the master will roll back the related operations.

V. EXPERIMENT

We measured performance on a WFMS cluster consisting of one master, 10 data servers and 16 clients. All the machines are configured with dual 2.7 GHz Intel Pentium G630 processors, 4GB memory, three 2 TB 5400 rpm disks, and a 100 Mbps Ethernet connection to Cisco SG200-26 switch. The 16 clients are connected to the other switch and the two switches are connected with a 1 Gbps link.

A. Reads

N clients read simultaneously from the file system. Each client reads a randomly selected chunk of different block. In order to compare the performances between small and large files, we set chunk size to 16KB, 128KB, 1M, 4M, 64M separately.

Figure 8 shows the aggregate read rate for N clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100Mbps network interface get saturated. The read rate for one client is 10.67 MB/s when chunk size is 64 MB, while the rate is 445 KB/s when chunk size is 16 KB. The small chunks have lower rate, it's because lots of time consumed on the network connections. As a whole, the performance of WFMS is satisfying and it behaves well in practice.

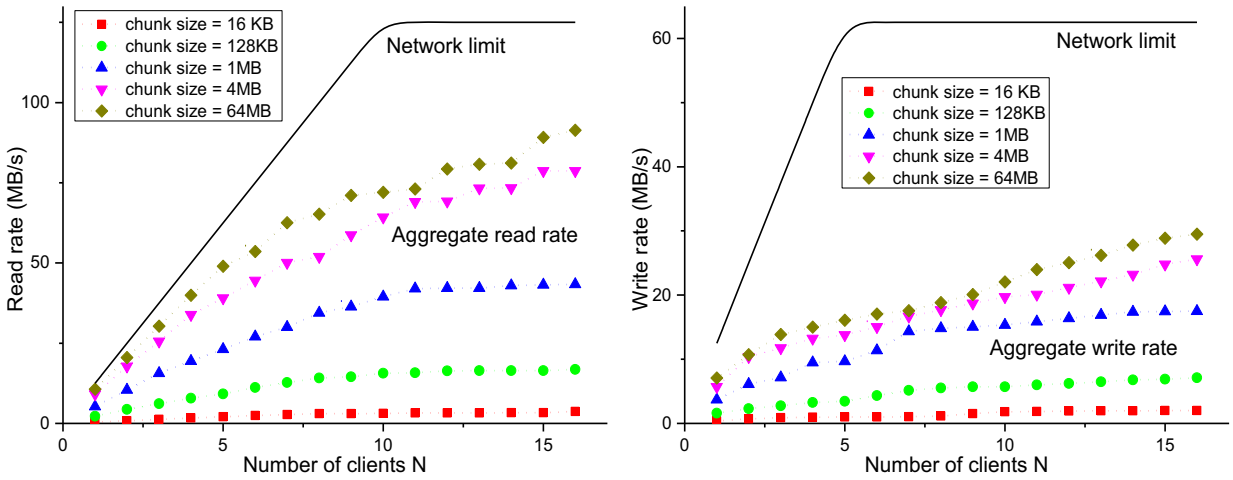

Figure 8. Aggregate Throughputs.

B. *Writes*

N clients write simultaneously to N distinct files. In order to compare the performances between small and large files, we set file size to 16KB, 128KB, 1M, 4M, 64M separately.

Figure 8 shows the aggregate write rate for N clients and its theoretical limit. The write rate for one client is 7.08 MB/s when chunk size is 64MB, while the rate is 394 KB/s when chunk size if 16 KB. The small chunks have lower rate, it's also because of the network connections.

C. *Replicas*

In our system, the number of the hot block copies will be increased dynamically shown before. In order to show the effect of this, we use 16 clients to read one same block frequently. Each client read once per 30 seconds, and this process last for 400 minutes.

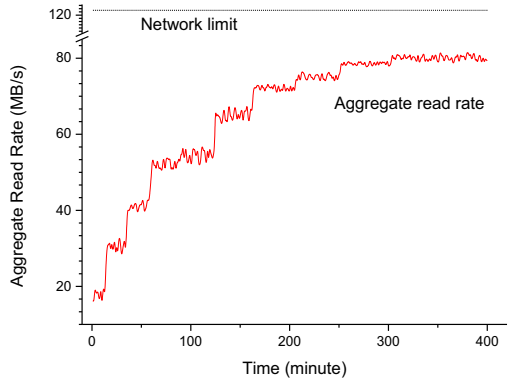

Figure 9. Frequent Reading of One Block

Figure 9 shows the aggregate read rate for 16 clients and its theoretical limit. We can see the aggregate read rate is 16.25 MB/s at the beginning with the default two replicas. With the pass of frequent reading, the system create more replicas, which results in the improving the aggregate read rate.

VI. CONCLUSIONS AND FUTURE WORK

In conclusion, WFMS is a distributed file system providing storage and management service. It has single master and multiple data servers. The block/chunk mechanism makes WFMS well support both big files and small files, with high space utilizations. The hot block mechanism greatly boosts the reading speed of frequently read files. In the experiment and the practical application, WFMS performs very well.

WFMS is in developing. There's lots of work to do. The mechanism to handle component failure will be improved. Image/Journal mechanism of HDFS may be used for recovery. How to distributed block is also worth studying next.

REFERENCES

[1] K. McKusick and S. Quinlan, "GFS: evolution on fast-forward," Commun. ACM, vol. 53, pp. 42-49, 2010.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, 2010, pp. 1-10.

[3] R. B. Ross and R. Thakur, "PVFS: A parallel file system for Linux clusters," in in Proceedings of the 4th Annual Linux Showcase and Conference, 2000, pp. 391-430.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," presented at the Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 2003.

[5] D. Borthakur, "The hadoop distributed file system: Architecture and design," Hadoop Project Website, vol. 11, p. 21, 2007.

[6] www.taobao.com

[7] http://wiki.apache.org/cassandra/ArchitectureOverview

[8] Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels. Dynamo: amazon's highly available key-value store.SOSP2007.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et,al..Bigtable: A Distributed Storage System for Structured Data. Proceedings of OSDI'06: Seventh Symposium on Operating System Design and Implementation, 2006:205-218.