MFS: A Lightweight Block-Level Local Mirror of Remote File System

Chunxia Zhang
College of Electronic Information and Automation, Tianjin Science & Technology University, Tianjin, China
Email: zhangchx@tust.edu.cn

Baoxu Shi and Xudong Li[1]
Software College, Nankai University, Tianjin, China
Email: {shibx, leexudong}@nankai.edu.cn

***Abstract*—Nowadays individual users often have more than one computing platform, such as traditional desktop in office, laptop computers in home, and mobile smartphones in outdoor. More and more users store their data into remote file system and access them over network in every time, but user has to download the whole file to the user computer before user wants to access one file, so user has to have the poor experience, especially access large file over wireless network. We have developed MFS, which is a lightweight client-side local mirror of remote file system. MFS mainly provides four mechanisms to solve the above problems. One mechanism is that MFS uses client-side file system based on disk as a persistent cache for files, and the capacity of the persistent cache is limited. The second mechanism is that MFS uses block-level granularity as the smallest unit of file access operations and transmission. The third mechanism is that taking event publish–subscribe pattern to keep files system consistent between user client and remote network file system server. The fourth mechanism is that taking different file consistency priority strategies for different types of files. All files will be stored on cloud or remote file system, but only some files which are often accessed recently will be stored on user client-side persistent disk transparently. So user can have a larger logical storage space than user local disk, and user also gets high accessing speed of accessing remote file system, which speed is close to the speed of accessing local disk file system. User's applications can always access files, and do not wait until all the blocks of the file is downloaded. Our evaluation demonstrates that MFS has a good performance, reliability, transparent scalability and simplicity. MFS can run on a diversity of user computers, and it is independent of any computer.**

***Index Terms*—file system, persistent cache, block-level, file consistency, publish–subscribe pattern**

I. INTRODUCTION

Personal users have to deal with many computing devices every day, so users have to face the problem which is how to share files among their computers. A simple solution is that copying files from one computer to another computer, but it will often cause files inconsistencies. With the growing trend of Internet, especially the raise of cloud computing and wireless network, more and more users will consider to store most of their data on the cloud, which is depicted in Fig. 1. By storing on the cloud, two advantages are easily to find:

1. Cloud services could provide large data storage space in a more competitive price than local storage space.
2. Data stored on the cloud could be accessed over the internet by any computing devices (such as personal computer, mobile).

Nowadays, there are two kinds of user's data access modes. One is online mode, which means users or applications need to download the data from remote server to local storage media (memory or hard disks) before they could access it. Another is offline mode, which means users store the data they needed on local hard disk in advance, so network is not needed during user application runs.

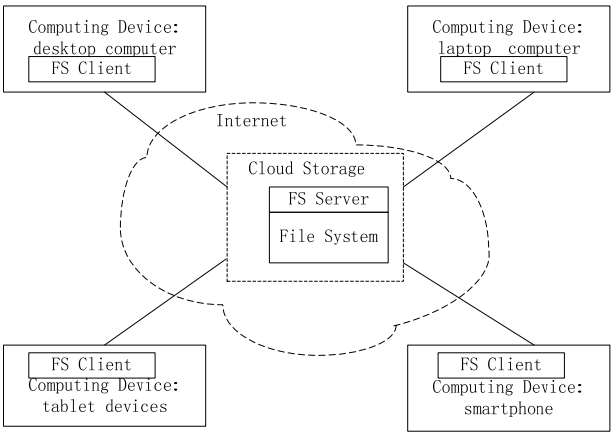

Figure 1. Personal User Computing Environment Architecture Based on Internet

The online mode could provide us unlimited storage, and files could be accessed by any computing devices in any location. Typical representatives of online mode include NFS[1], CIFS[2], FTPFS[3], SSHFS[4], HTTPFS [5], GFS[6], and HDFS[7]. But there are also some disadvantages:

1. The user computer must keep network online, if the network is an unreliable or even disconnected, user current work would be discard by network problems;

[1]Corresponding author: Li-Xudong

doi:10.4304/jsw.8.6.1459-1470

2. The user experience would by easily influenced by bandwidth and stability of networks, because the user computer is highly depended on internet connection conditions. Especially in wireless network, according to the size of files, user may suffer unbearable delays due to low-bandwidth and high latency.

In the offline mode, all files are stored in user local computer, so user uses files without network connections. So offline mode provides a good user experience. But there also have some disadvantages:

1. The disk space of user computer is limited, local storage space is much smaller than the network storage space. As the data size growing, increasing the number of local disks is difficult to unrestricted expansion;

2. These files are only in specific computing devices, so it would be hard if users want to access their files on other computing devices.

So a hybrid model was introduced, in which mode, users store files on the cloud, when they need to use some of the files, they download these files from cloud and store it on local disk before using it, but the network connection does not need in every time. P2P downloader (such as BT, eDonkey, Thunder) and network drivers (such as Dropbox, Microsoft SkyDrive, and Google DriveSync, Apple iCloud etc.,) are belonging to this mode. Typically, P2P downloader is only responsible for the download tasks of the distributed data, and does not maintain consistency problems, and Dropbox will automatically updates changes happened in target directories to keep consistent between user local computer and remote network file system.

Dropbox[8] also has some limits. One is that the shared directory of user computer is limited by the physical size of local disk storage. With the increase of data, users can not download all files of remote file system to their local disk storage. The second is that Dropbox provides file-level operations to user applications, which means if user wants to access a file, user need to wait until the file is downloaded from remote file system completely. The download process is not good enough to support a great user experience. In addition, this download model will increase unnecessary network communication overhead, especially in wireless network environment , because user maybe only read a small part of one large file and close it for a long-term time, though the upload process of Dropbox is based on binary diff and Incremental transmission. The third insufficient is that it is hard to keep file consistency among users' computers.

In users' point of view, the requirements of data storage have four main functions.

1. Users want to have an unlimited storage which is far larger than local storage on their devices.

2. Users want to have a good experience when accessing their file. That means easy to access, simple to use and simple, transparent management.

3. Users want to access their data from any computing devices they want, and will not be limited on some specific devices.

4. The data of user is stored in a safe and reliable environment.

There is no doubt that put data on the cloud could satisfy the requirement (1) and (3), and partially meet the (4)'s need. As we mentioned before, in online network mode, it would be hard to provide a good user experience, because they do not have a user-side persistent disk as a cache. Without that, each time users reboot the client or even disconnect from the server, they need to retrieve data again. That means the service's performance is based on network bandwidth, not the local disk bandwidth. In fact, the network bandwidth will not exceed local disk bandwidth (such as SAS, SSD) in the next period of time.

To improve users' experience, a hybrid model was introduced by AFS[9] and Coda file system [10]. Coda uses the local disk as a file cache, and also support disconnected operation. So Coda offers the good availability and performance, and also provides the high degree of consistency attainable within those constraints.

But Coda also still has some limits.

1. The capacity of Coda client persistent disk is limited by the physical size of local disk storage. In Coda, files and directories are cached in their entirety by clients. If user will access many files which total size is larger than the capacity of the local persistent disk, coda does not work normally.

2. The second is that user has to wait to access a file until the whole file will be downloaded to local device.

In this paper, we explore an approach for solving the above problems for personal user computing environment with multiple devices. We introduce a new lightweight local mirror of remote file system called MFS, which uses local disk in user-side as persistent cache.

MFS is different to traditional tightly coupled distributed file system, and all files of remote file system are not completely uniformly dispersed in many client computing nodes. In fact, MFS is a mirror of remote file system, and MFS is just a partial copy of remote file system. MFS mainly provides the following new mechanisms to solve the above problems:

One mechanism, using user client-side file system based on disk as a persistent cache for files, which capacity can be specified by user, i.e. it is limited. So MFS will not occupy all of the user's disk space.

The second mechanism is that using block-level granularity as the smallest unit of file access operations and transmission. Coda[10] doesn't support it.

The third mechanism is that taking event publish–subscribe pattern to keep files system consistent between user client and remote network file system server.

The fourth mechanism is that taking different file synchronization and consistency priority strategies for different types of files, so user will achieve better experience.

Based on the above mechanisms, all files will be stored on cloud or remote file system, but only some files' copies will be stored on user client-side persistent disk transparently. So user can have a larger logical storage space than user local disk, and the logical storage space is provided by remote file system. At the same time, user also access remote file system at a high speed, which speed is close to the speed of accessing local disk file

system, and far greater than the speed of the pure network online mode. User's applications can always access files, and do not wait until all the blocks of the file is downloaded. MFS could dynamically adjust file working set, and transparently retrieve file from remote file system. All operations of files on user computing device will be logged and uploaded to remote network file system to keep consistent using different priority of consistency for different file, so we present a loosely coupled distributed file system to particularly suitable for wireless network.

We present the design and implementation of MFS. The rest of this paper is structured as follows. Section 2 motivates the proposed architecture of MFS. Section 3 provides presents the core functionality of the system to resolve the above problems. Section 4 addresses some optimizations of MFS. Section 5 and 6 presents prototype implementation and evaluation. Section 7 reviews related work and Section 8 discusses related work. Finally, we present some concluding remarks.

II. SYSTEM ARCHITECTURE

A. Physical Topology Architecture

We present the improved architecture of personal user computing environment based on Internet storage. All files of User are stored on remote file system, and only small files' copies of file working set are dynamically stored in user computing device. If a file is modified, the updated file can be temporarily stored in user-side.

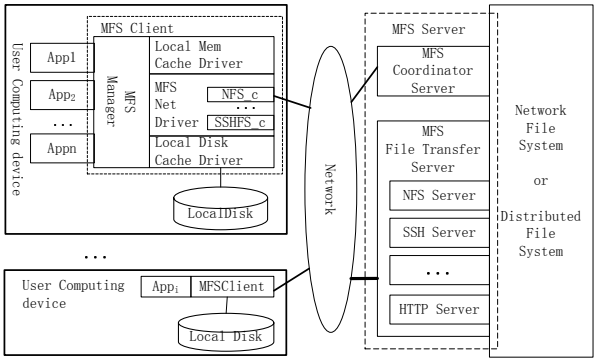

Figure 2. MFS Physical Architecture

The remote file system can be a single file system based on one computer, and can also be a distributed file system based on multiple computers over Internet, especially be based on cloud storage. We recommend the remote file system uses the tightly coupled distributed file system, because the file system can provide high reliability, availability, scalability, and consistency. Because a file can be modified by user, the latest version of the file may be in the user computer, and not in the remote file system. So in the new personal user computing environment, multiple copies in different users' devices need to be kept consistent. We recommend that all the files in multi-users computing devices and remote file system together to form a loosely coupled distributed system, which takes a block-level eventual consistency mode so only the users that cache the file are informed to update their related copies.

Fig. 2 shows the physical architecture of MFS. MFS has four layers: 1) user applications on user computer; 2) MFS client in user computer; 3) MFS server on Internet; 4) file system in Cloud storage. MFS Server provides block-level file system services for MFS clients. In fact, MFS client acts as the remote file system agent for the applications in user computer to provide file system services.

MFS client in user computer is responsible for storing a copy of recent used data blocks in a specific period of time at local disk device, and uploading the modified copies. MFS client uses both memory and local disk as cache. MFS client keeps these copies consistent with remote file system transparently, and also minimize the dependent of network and network cost.

MFS Server is involved in many common network protocols, like SSH, NFS, HTTP, CIFS, and FTP, so MFS client could be connected to MFS Server and Remote file system by the above network protocols. MFS Coordinator Server as an important component of MFS Server is mainly responsible for consistent event publish–subscribe.

MFS client uses virtual file system to provide file system services for user applications, so user applications could access the file system services of MFS without changing their codes. MFS is transparent to user applications just like other local file systems.

Because all the files of user are stored in the remote file system and the modified blocks of files will be updated to the remote file system in time, so user will no longer depend on a specific physical device to store these files, and no longer worry about these files are missing or corrupt. User could use any computing devices (such as personal computers, laptops, smart phones, enterprise servers) at any time to access their files by MFS.

B. Theoretical Cornerstones

1) Principle of Locality

The most important program property that we regularly exploit is the principle of locality. The locality principle of file access reveals an implication. The data blocks of files caches on local devices are just a portion of the whole data blocks of the remote file system. Because these caches are created with the use of locality principle, it could provide normal access to some files in file working set or partial data of a file in a period of time. Normal access means user can do operations on files correctly even if there is no network connection available or suffering a high latency.

2) Block-level Data Access Model

MFS uses block-level data model instead of file-level Data Model, by doing that, MFS could support caching partial blocks of data on local device. So we can split a file to several blocks, and also retrieve file data from different sources concurrently.

Fig. 3 shows a general case about the layout of file data block copies between user computer and the remote file system. All data block copies of files are stored in the remote file system, and only some small data block

copies of files are stored in the user computer. The latter may be in the memory of user computer, or in the disk of user computer, or in both the memory and disk of user computer.

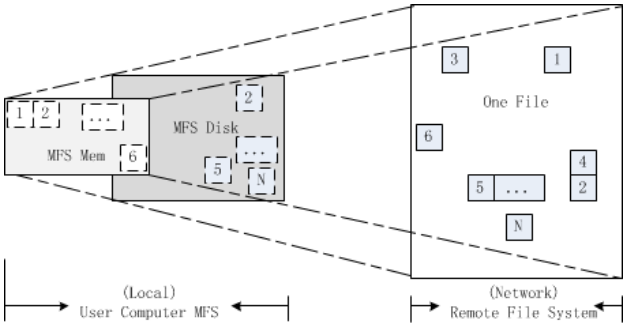

Figure 3. File Mapping of MFS and Remote File System

In MFS client, local disk storage will persistently store file working set and recent access data blocks in a period of time, so most data blocks of files which will be accessed by user has been cached on local storage media, but if user arise some request of files which copy is only in the remote file system, MFS client will dynamically update local file working set.

C. Logic Architecture of MFS Client

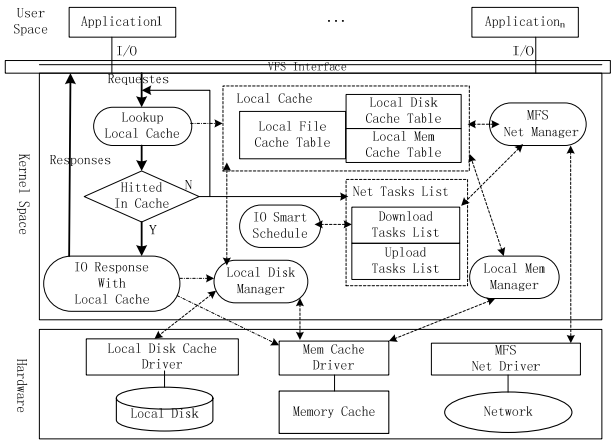

Figure 4. Logical Architecture of MFS Client

Fig. 4 represents kernel components of MFS client and its logical relationship and functions. File attributes and directory data are Meta Data of each file, file contents are Data of each file, both meta-data and normal data will be partly stored on local disk device.

MFS client uses local hard disk to persistently store a section of files cache, and uses local memory to provide high-performance data access. Persistent cache on local disk is a portion of remote file system, and temporary cache on memory is a portion of local disk's persistent cache. But temporary cache also could include data blocks come directly from remote file system when there is no cache of some new opened file which is not in current file working set.

Meta data and normal data have different cache area in both memory and disk. In memory, they are placed in two different memory regions; on the disk, they are split into small files, each of those files contains parts of the data. All these data are indexed by a MFS Global Controller. With this hierarchical index, MFS could manage and cache partial data blocks and meta data on user's local computer.

Local memory manager is responsible for manage temporary cache in memory. Because MFS could set the maximum size of memory cache, when temporary cache is close to predefined threshold, memory manager will use policies like LRU to drop out-of-time data blocks to disk, if such data blocks are modified by user, memory manager will save these blocks to Local disk cache by both synchronous and asynchronous method. After such clean operation, it will retrieve new data blocks to memory according to user recent IO access method.

Local disk manager manages data blocks of File Working Set on local disk. Because MFS could set the maximum size of the cache, when the size of disk cache is close to predefined threshold, disk manager will use strategies like LRU to drop unused blocks which is unmodified or already uploaded to the cloud.

I/O Smart Schedule take charge of creating and scheduling upload and download tasks and changing the priority of other threads by applied strategies, users' recent IO actions, count of modified data blocks and usage of memory and disk cache.

MFS net manager do upload and download tasks. It will check system load and decide either using synchronous or asynchronous method to do the tasks it received.

The standard MFS read process is as follows. When MFS get a read request from user, it will assign a response thread, which will check the block map of the file to locate the request data blocks. If data is cached in memory, response thread will return the data blocks instantly, if it is cached on the disk, response thread will retrieve the data from local disk and put an asynchronous task to update the memory cache. If data blocks are not cached, response thread will create a data request task to I/O Smart Schedule, and blocked at waiting list until the data is get from cloud. If MFS find request data is not stored in memory, it will create read ahead tasks to retrieve data from cloud or local disk in order to increase the effective and efficient of IO access.

The standard MFS client write process is as follows. When MFS get a write request from user, it will assign a response thread. This thread will check the block map of the file to locate the request blocks. If blocks are cached on local devices, the thread will overwrite the corresponding data blocks. If it is not cached, the thread will create a data request task to I/O Smart Schedule, and wait until data retrieved, then overwrite the blocks. If this file is not exists on local device or cloud, responding thread will create a new inode and allocate block maps of it, then write data blocks and update the inode. After these operations, return the write bytes or related error code to user.

In order to describe the simple, the above processes is not taken into account the consistency, the next section focus on how to keep the consistency of files.

D. Block-level Eventual Consistency Model

MFS take a block-level eventual consistency model. All operations of files on user computing device will be logged and uploaded to remote network file system asynchronously.

MFS coordinator server is responsible for consistent event publish–subscribe.

When a MFS client of user reads some blocks of files from remote file system, all the information include file and block number ID are recorded by MFS coordinator server, so the latter will know where a block copy of file among multiple users devices is.

When a MFS client of user want to modify a file, the file firstly is locked with local lock by the MFS client synchronously, then MFS client applies to get a global lock for the file from MFS coordinator server in time. If the file was not global locked before, MFS coordinator server will assign a global lock for the file with the MFS client ID. So the MFS client will acquire a global lock for the file asynchronously.

While the file is locked with global lock, if another user computer wants to apply the global lock, the applying operation will be failed and the latter just only

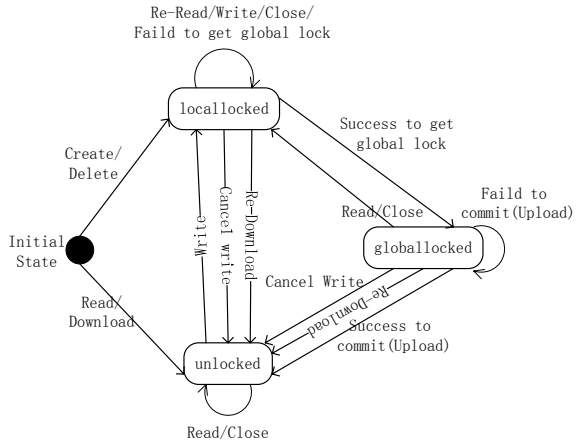

Figure 5. File State Chart

gets the local lock.

When the MFS client has successfully committed the modified file with global lock to remote network file system, MFS coordinator server will notify all the MFS clients caching the block of the modified file to the update news asynchronously. The news will at least include the filename and block num, so all the MFS clients caching the block of the modified file will only set the related block is invalid.

The file state transition is depicted in Fig.5.

MFS client also supports to upload the modified files using different priority of consistency for different file. Such as object files and temporary files created by user applications will be uploaded to remote file system at last, in particular, MFS client won't even upload these files.

III. DETAIL DESIGN

A. Key Data Structures

MFS client use Global Controller to manage temporary and persistent cache of partial files in remote file system, MFS current status and controller information.

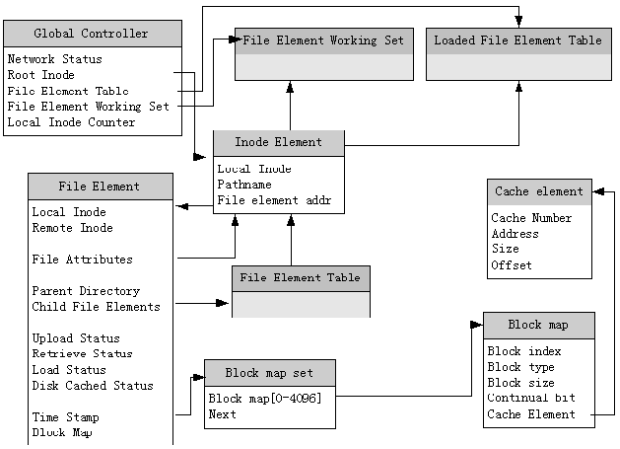

Figure 6. MFS Key Data Structure

The key data structure of MFS is depicted in Fig.6.

Each cached file (including directory) has a unique Inode Element, it contains local Inode ID, corresponding filename and pathname. This structure also points to a File Element, which provides detail information, like data block status and system based attributes about this very file.

Each normal file cached in local devices use small portion based storage, which save data discretely on memory and disk. MFS use Block map to index each Cache Elements of files.

Each cached directory contains at least one File Element Table which has the information of related sub files' index information.

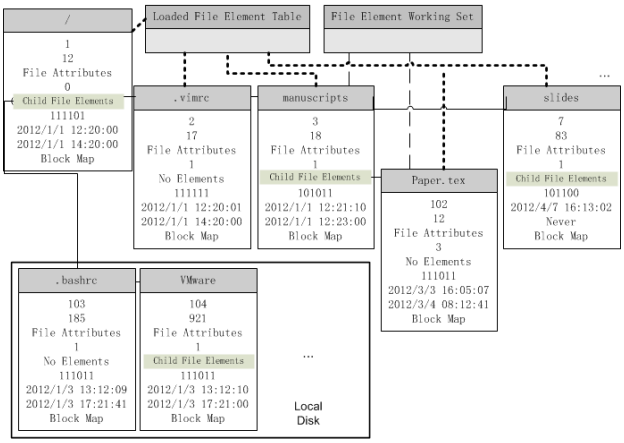

Figure 7. An Example of File Searching and Indexing

To improve MFS's performance on indexing files, MFS user File Element Working Set and Loaded File Element Table to provide fast index of recent used files. If loaded file element becomes larger, MFS also provide the classic hierarchical index for file searching.

Fig. 7 shows an example of file searching and indexing structures. The root contains a .vimrc file and two

directories, manuscripts and slides. Manuscripts directory contains a Paper.tex file.

If we check /manuscripts/Paper.tex, the File Element will be returned when checking File Element Working Set. If we check /.vimrc in MFS, firstly it will check File Element Working Set, because /.vimrc does not exists in this table, MFS will check Loaded File Element table or searching this file by hierarchical index according to the size of loaded elements. If MFS check file by hierarchical index, it will get root (/) first, then searching in the root's Child File Element table to locate the target File Element. If we search /.bashrc, after checking File Element Working Set and Loaded File Element Table, MFS will get root directory and search for Element, obviously it returns a empty result because the File Element of /.bashrc is located on the disk. By checking Loaded bit of root, MFS find the sub files of root are not all loaded, so it will check the hard disk cache then insert this into memory cache, and finally return the File Element.

When memory cache or disk cache's size meets the threshold, MFS uses LRU algorithm to replace unused data blocks in order to recovery usable cache space.

B. Block Map Based File Retrieving

Fig.8 shows an example of file reading with block map. In this very block map, first 30 blocks are remote blocks, which mean MFS do not have a cache on these blocks. Block 31 is a hole block, block 32 to 63 cached in memory, and block 64 to 71 is cached on disk.

If reading request wants to get a bunch of data sized

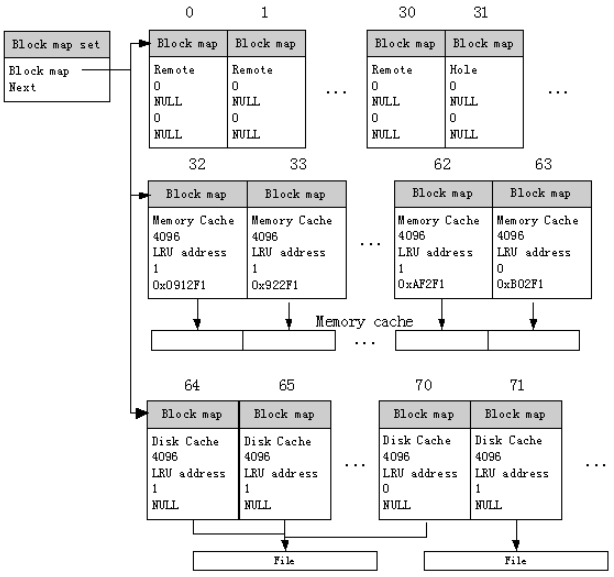

Figure 8. Block Map Based File Retrieving In MFS

131072 bytes from 0, MFS will look up the block map and find out that first 30 blocks are remote blocks and the last block is a hole, so it create a data request task and return the data 131072bytes with the last 4096 bytes is zero.

If reading request wants to get 40960 bytes data started at 25392 bytes (block 62), then MFS will check the block map and find out block 62 and 63 are continual stored data blocks, so MFS will get these two block in a single operation. And then it finds out that block 64 to block 60 is continual stored on disk, so MFS also combined reading operations to a single one. Finally MFS get the last 71 block from disk and construct the return data which is 40960bytes.

If MFS got a write request to write 40960 bytes data started at 25392 bytes (block 62), firstly MFS will mark these blocks as dirty blocks to avoid unintentional deletion, and then it will overwrite block 62 and block 63 and save them to disk, and over write block 60 to 64 at once, then write block 71. The log of this writing will send to I/O Smart Schedule, when upload task is finished, the dirty bit will be set to zero.

When MFS is offline and I/O request contains non-exist data blocks in cache, MFS will record the unfinished operation to log file and return an EAGAIN error, otherwise it will return the data correctly. When a previously offline MFS client comes back online, all unfinished operation will be redo.

C. Local Memory Cache Management

The size of MFS's local cache is user defined. That means MFS could be run on devices with small storage by adjusting its cache size to a smaller one.

Because MFS could control data at block level, we create a small size memory cache to load recent used blocks of File Working Set to accelerate I/O speed by increase cache hit rate in memory.

To achieve this, MFS need to keep recent used data blocks stays in memory cache and release unused blocks to save memory for new data blocks. Recently, MFS use a classical LRU strategy to manage memory data blocks. There is a cleanup thread refresh it periodically.

MFS also measures which blocks could be add to memory cache. Firstly, the block should belong to current File Working Set. In addition, the block should be read by I/O request in a short time after added.

So, a block belongs to a pdf file in a File Working Set could be put into memory cache, but blocks belongs to a AVI could not, because it may not be accessed instantly.

D. Local Disk Cache Management

MFS's disk cache does not use the normal file-level cache, instead, we decide to use a mixed method, and a block based cache stored in small files.

With this approach, MFS could only save small portion of files instead of entire file. And that could make cache on local device become more effective and efficient especially when cache size is limited to a relatively small size. To be specific, think about Virtual Hard Disk Images, although user's client may not have enough space to store such a larger file, but MFS's disk cache could save the necessary data blocks which is necessary for VM's to start up, and provide a local access experience when user using it. MFS also use the same LRU strategy to update caches.

E. Network Management

MFS Net Manager takes charge of operate upload and download tasks by monitoring system loads. When the system load is low, it will run both download and upload tasks, when the system load is close to predefined

threshold, some synchronous jobs will be set to asynchronous, non-important read ahead and upload tasks will be prevented.

1) Download task

Download tasks types include instant read, in time read ahead and user action based read ahead.

Instant read is the task that created by I/O requests and should never be delayed. In time read ahead is created by user's reading operations, if user is reading a file in a specific mode, like sequence reading, in time read ahead tasks will be created to in order to shorter I/O response time. User action based read ahead is created when user access a directory, this task is the lowest priority downloading task, it will execute only if the system is idle. And the data it retrieves will be written to disk cache directly and add to LRU randomly.

MFS takes auto asynchronous download ahead with multi-polices.

As we mentioned on last section, MFS has two Read ahead levels, in time read ahead and user action based read ahead. In time read ahead is based on I/O requests, and user action based read ahead is, as its name reveals, based on user's access habits.

In time read ahead is raised when read request is received. In MFS, we do not try to pre-download the block of data right next to current request. As an alternative, we use a stride-based read ahead to reduce duplicate download task numbers and improve reading performance.

The Instant read ahead work flow is as follows. Firstly it was created by an I/O request, then put this task into corresponding task list. When executing these tasks, MFS must check the valid of target data blocks on local cache, if such cache already exists, this task would be dropped. If it is valid, do this task and put data blocks to memory cache.

User action based read ahead, on the other hand, is raised when user's data access method meets some of the predefined strategies.

For example, the simplest strategy is that when user enters a directory, MFS will create read ahead tasks for each element in this directory. And if user enters a directory contains lots of media files, the read ahead task will not be created because such data would large and mostly useless.

2) Upload task

Upload task types include instant upload, in time upload and idle upload.

Instant upload is a synchronous operation, which need to upload the modified data blocks or meta data immediately. Instant upload is an asynchronous task, it will upload the related data when the system load is not too high. Idle upload is an asynchronous task with lowest-priority. It will be executed only if the system is idle.

MFS also takes auto asynchronous upload ahead with multi-polices.

In MFS, all the data modify operations, like create, edit and delete files will be performed at local cache. Each operation will create an upload task. These tasks will be uploaded by Network Management and execute on the remote file system.

The three upload mode mentioned before have several execution strategies.

Instant upload: To keep the file consistency, when data is changed at local devices, MFS will send the editing message and data block to remote file system instantly. When MFS is offline, instant upload task will turn into asynchronous mode to avoid user application blocking. When connection is reestablished, it will uploaded immediately and switch back to synchronous mode.

In time upload: MFS promise such upload task will be executed as soon as possible. When the system load is not too high and the execution of these tasks cannot influence the performance of local device, these tasks will be implemented. This method suits for people who do not have a highly consistent need or using a device with limited performance.

Idle upload: MFS will delay the upload task and only upload it when local device is idle. The typical use of this method is when local cache is full, and there are lots of dirty blocks need to be dropped. In such condition, MFS will use temporary extra disk cache to store these dirty blocks and add idle upload tasks to transport it.

F. I/O Smart Scheduler

I/O Smart Schedule responses for scheduling upload and download tasks and changing priority of threads by calculating system loads and I/O access methods.

As mentioned in Fig. 9, I/O Smart Scheduler provide a Public Watcher structure to other threads, and it update this periodically by monitoring CPU, I/O and Network usages.

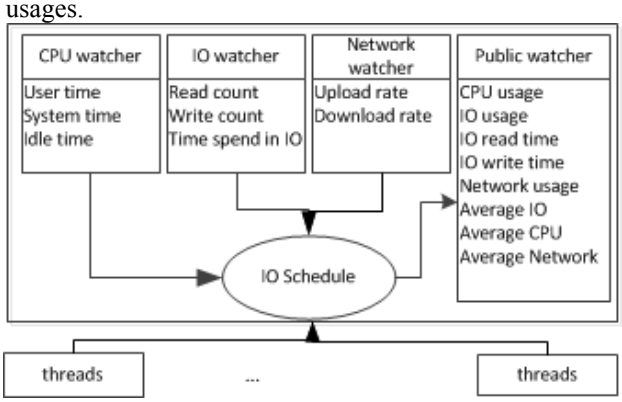

Figure 9. I/O Smart Schedule Architecture

To calculate CPU usage, MFS check user time, system time, and idle time of client's CPU, by evaluate the percentage of idle time in sampling period, MFS could get a CPU work load. To calculate I/O work load, MFS check I/O operations¡¯ count and the time it consumed. The percentage of I/O average operation time and the ratio between executing time and idle time in sampling period, could reveal I/O status of the client. By checking the network sending and receiving bytes per sampling period, I/O Smart Schedule could get the current usage of network. The average values of system work load may reset when one of the CPU, I/O or network status meets a threshold to keep it valid. Without that, the average may become higher than its real performance.

I/O Smart Scheduler uses an Observer strategy to send system loads and priorities to other threads. Other threads which registered to I/O Smart Schedule, will check Public Watcher every time they start an operation, by comparing average workload and current workload they will adjust their priority of execution, and decide whether block themselves or keep on running. This policy minimize the work that done by IO Smart Schedule. In that way, the sampling period would be more precisely than imitative blocking, because in such mechanism, if IO Smart Schedule tries to block a thread, it may need to wait until an atomic operation finish.

IV. OPTIMIZATIONS

The MFS introduced in the following several optimization strategies, effectively improving the MFS online performance and user experience.

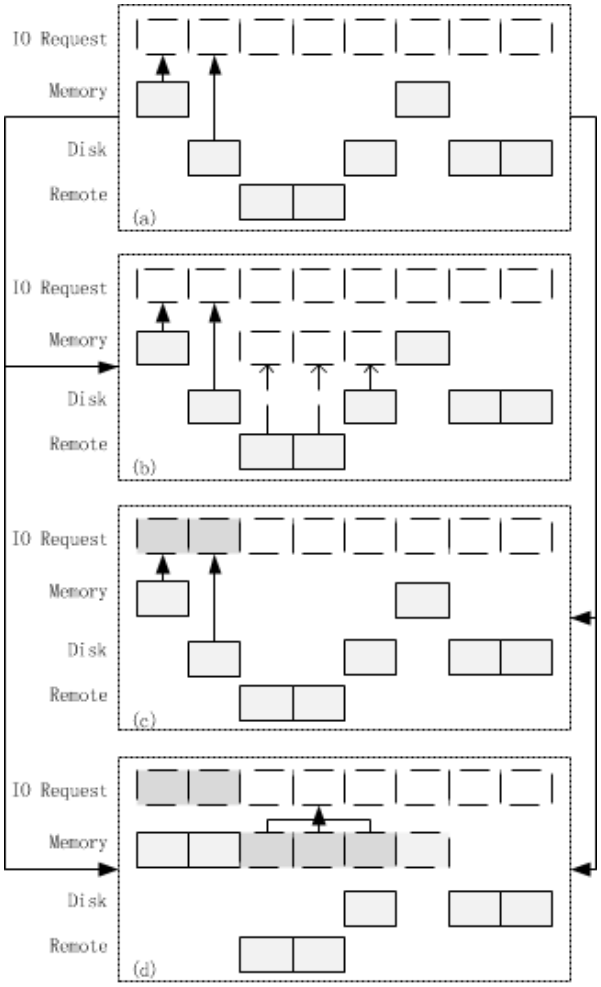

Figure 10. Read Ahead from Local Disk and Remote File System

A. Read Ahead Technology

Fig.10 shows the concurrent read ahead and cache clean up mechanism.

In (a), MFS got a read request, but because of inconsistent, it need read twice to get the data. During the implementation of this read request (c), MFS is concurrently reconstructing the memory cache.

In (b), MFS is doing read ahead and memory reconstruction. When read request is received and system load is not meet the threshold, MFS will retrieve data from remote or disk to cache in order to speed up I/O performance. As we see, during operating the read request, MFS copy remote and disk cache to memory which is adjacent to existent caches.

In (d), when read request is accomplished, the following three blocks are already in memory. And the next read request will only need to do a single read then it will get all three blocks of data it needed.

B. Dynamic I/O Load Control Technology

MFS has a build-in system work load controller, which could avoid MFS use too many CPU time, memory caches, and also bandwidth.

Unlike SSHFS which build its cache entirely in client's memory, MFS use a user defined, fixed size memory cache as a swap, which storing data blocks of File Working Set, and using a periodical clean up thread to make sure data in memory is not exceed specified size. As a tradeoff, we store most of the caches on client's local disk, there is also a cleanup thread monitor on it, but for consistent reasons, when the disk is full of dirty blocks and internet connection is disabled, the size of disk cache may exceed.

Except controlling the usage of storage media, MFS also use a three-level download and upload strategy to avoid the overuse of bandwidth and local I/O. Each time MFS do a network task, it will check the status send by I/O Smart Schedule to find if they need to be blocked until network condition is stable or less busy. As mentioned before, MFS's read ahead will not only retrieve data from remote server but also from local disk cache. Such implementation may raise a lot of local I/O requests especially when reading a large file which is cached. At the worst case when all the caches are belongs to one single file (To be more specific, that may be a Virtual Machine's virtual hard disk or a blue ray disk image), in this very condition, read ahead will load all the cached blocks on the disk to memory cache and write current memory cache back to hard disk. Such method may cause a huge amount of unnecessary I/O operations and significantly slow down the client device. To avoid that, I/O Smart Schedule will calculate the I/O requests per sampling period and the executing time of each I/O request. When either of them is higher than average value of this current MFS execution, MFS will reduce the concurrent read ahead on the disk, when they meet the threshold, MFS will disable read ahead on the disk. Because local file systems also have cache mechanism, the consuming time is not highly increased, but by doing that, we could reduce the side-effect of MFS cache and

read ahead system which may cause high latency of user experience on other applications.

V. IMPLEMENTTATION

MFS can be implementation by FUSE, VFS, WrapFS and other file system framework. FUSE is a popular framework in user space of operating system. This paper recommends FUSE-based MFS. Our MFS uses Glib to apply thread, memory and data structure implementations, and using XML to manage configure files. MFS is developed on Linux by C. The core program consists of 15000 lines of codes.

MFS could be built on any UNIX-like system, for example, Solaris, BSD, etc., which has a VFS layer. Also, MFS could be done on Windows with IFS DDK or SFilter.

VI. EVALUATION

This section evaluates the performance of MFS in both benchmark tests and real usage environments. We use iozone as the benchmark tool, and the real usage test we use kernel compiling, running Virtual Machine on a PC with limited hard disk storage and other everyday operations.

The following performance test is done with two computers and a wireless network connection. The Client is a Intel CORE i5 quad-core 2.4GHz DELL laptop with 4G memory and 20G hard disk. We take iozone software as the test tool.

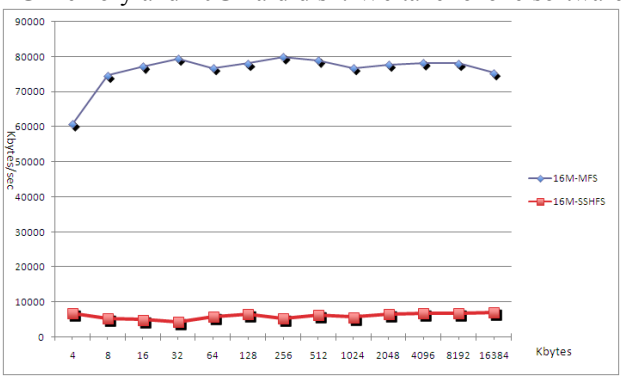

Figure 11. MFS vs. SSHFS in Sequential write

A. Sequential Write

Fig.11 measures the performance of rapid sequence write. Here we only choose a 16MB file as the test sample.

Because MFS has an asynchronous upload mechanism, which delayed upload operations if client's I/O and operations are busy, it has a better performance on writing test. While SSHFS synchronously upload its data, the result of these tests is much lower than MFS's.

It is easy to make a conclusion that reduce and delay network uploads could effectively improve writing performance, because it releases CPU and IO time from operating such tasks.

B. Sequential Re-Write

Fig.12 measures the performance of rapid sequence rewrite. The combine and delay of upload tasks also contributes to these results. But the results of re-write are same as first time write, no matter it is MFS or SSHFS. To improve this, we need to add the local write cache mentioned before. By this, we could change disk I/O speed to memory I/O speed.

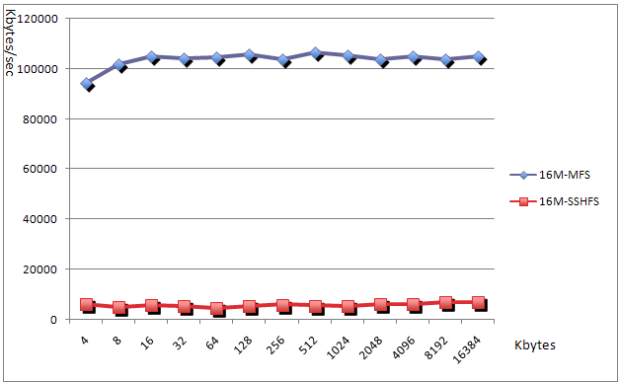

Figure 12. MFS vs. SSHFS in Sequential Re-Write

Just like write performance test, MFS is better than SSHFS because we add a delayed upload strategy to balance I/O operations and CPU usage.

C. Random Write

Fig.13 measures the random write performance. MFS is also better than SSHFS due to its late update strategy, and because of random writing does change block maps rapidly, its performance is better than normal sequence writing.

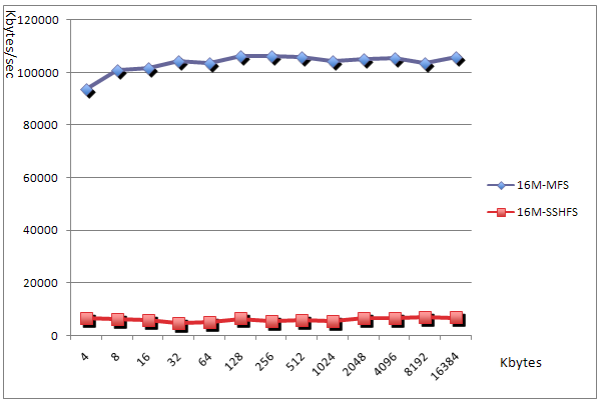

Figure 13. MFS vs. SSHFS in Sequential Random Write

D. Sequential Read

Fig.14 measures the sequence read performance. The temporary file that Iozone read is written by pervious writing test, which means MFS have some file cached at local disk and memory, while SSHFS may not have disk cache.

Because MFS has stride read ahead mechanism, our read result is better than SSHFS.

SSHFS drop caches by a timer (the default out-of-date time is 20 seconds), when the data size become bigger, the retrieve time is also increasing. So the cache of it will become ineffective, especially in a wireless environment.

But MFS use a much more effective strategy, the deletion of caches is based on its last used time, in that way MFS could have a higher hit rate than just based on timer.

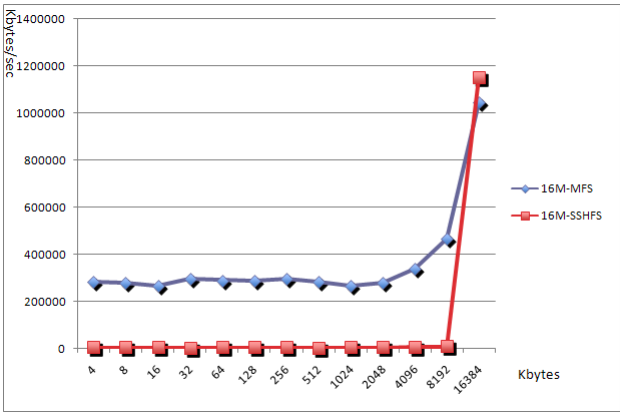

Figure 14. MFS vs. SSHFS in Sequential Read

E. *Stride Read*

Fig.15 measures the stride read performance. Because MFS uses stride based prefetching, so when reading operations are also stride, it is normal MFS got a higher score than normal read.

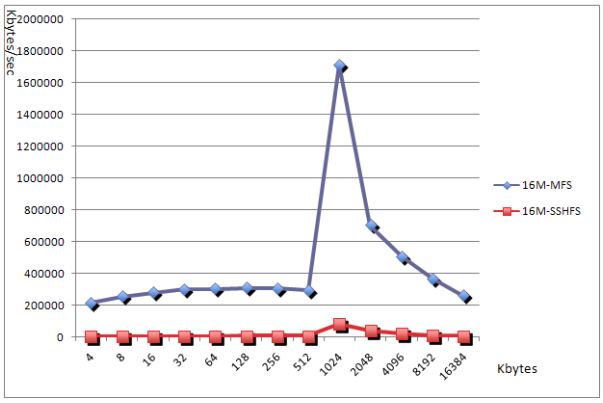

Figure 15. MFS vs. SSHFS in Stride Read

F. *Large file Read and Re-Read*

Sometimes, user's client has not enough memory or disk space, so we use a VM with 1G RAM and 660M free disk space to test file system performance on a limited computing resource.

We use a 700MBytes file to test performance on sequence reading of a large file. MFS uses 32.062s to load the entire file, meanwhile, SSHFS needs 42.760s, that due to our prefetching strategy, and when testing reread performance, MFS uses only 8.549s on average, but SSHFS still needs 40.8585s.

This test result shows that when the client resource is limited, SSHFS cannot cached file correctly, but because MFS has a block-based cache system, it could still cache file when client do not have sufficient memory or disk space. This feature is necessary when user wants to run a large file regularly, for example, run a VM on a netbook.

G. *Compiling Linux Kernel*

TABLE I.
COMPILING LINUX-2.6.35.13.TAR.BZ2 KERNEL

	MFS	SSHFS	Local FS (Ext4)
Compile Time(second)	412.927	3197.425	212.39

Table 1 contains kernel compiling time on different kinds of file systems. The performance of MFS is superior to SSH, and it's near to the performance of Local FS.

VII. RELATED WORK

This section describes the work related to MFS.

NFS[1], SSHFS[4], CIFS[2],FTPFS[3] are all belong to network online mode. These network file system generally will create a temporary cache in the user's computer memory, all the cache will be invalidated when the user restart the computer. When the network is unstable or more slowly in the user experience will be very poor.

DropBox[8], GoogleDriveSync, SkyDrive, and etc. are all belong to improved network offline mode provide a local directory with remote directory synchronization, but the local directory of user computer is limited by the physical size of local disk storage. And they only provide file-level operations to user applications.

As mentioned before, AFS[9] and CODA[10-14] file system also uses client cache to provide high performance and high consistency, so all clients computers consist of a tight coupled distributed file system.

BlueSky[15] uses a proxy server caching remote data to provide enterprise internal computers. But it cannot provide lower-latency responses between user computers and proxy server. LBFS[16] studies similarities between files or versions of the same file. If the same data is in client's cache, it needn't send these data. SafeStore[17] presents a distributed storage system designed to maintain long-term data durability, which is based on fault isolation. Sprite[18] uses large main-memory disk block caches to achieve high performance in its file system. Hejtmánek[19] studies a lock-free distributed data storage framework based on versioned files. Smaldone[20] focuses on Securing access among mobile computing, and proposes a Working Set-Based Access Control to restrict network file system accesses from untrusted devices. Rosenblum[21] presents a log-structured file system, and writes all file system changes to a log-like structure on disk. Ju[22] discusses the key technologies in cloud-based storage and distributed file system.

VIII. FUTURE WORK

As future work, we will focus on the new polices of read ahead to improve the read performance. To solve the security problem about network file systems, additional

encrypt module could be added to encode file blocks. In that way all data except meta data and directory structure could be encoded on the cloud. To keep data safety, we could introduce file system checkpoints to improve our service's reliable and consistency. To reduce network usage and improve efficiency of communication, MFS could compare redundancy of blocks between local cache and remote file systems to cut off unnecessary transportations. For mobile computing, the energy is a big problem, so we will improve the MFS with energy-related optimizations.

MFS could introduce multi-version control to support multiple accesses and automatically solve the file conflict problem. MFS also could introduce P2P to speed its performance, especially sequential read.

CONCLUSIONS

In this work, we present a MFS, which is a lightweight client-side shadow mirror of remote network file system. MFS uses local limited disk as persistent cache, block-level granularity as the smallest unit of file access operations and transmission, event publish–subscribe pattern to keep files system consistent, and different file consistency priority strategies for different types of files. MFS has large logic size which is same with remote file system, and has smaller physical size than remote network file system, and has good performance which is nearly local file system speed. We describe the MFS architecture and key structures, and also discussed some of the optimizations which are necessary to achieve good performance and low network cost. In MFS, files of file working set are stored on user computer disk, user can view and access all files of remote file system by MFS with high speed as the local disk file access. MFS also support disconnected operation. MFS can be portable to different computers. Our evaluation demonstrates that MFS is Effectiveness. Overall, we believe that MFS is promising in individual user computing environment based on Internet storage.

ACKNOWLEDGMENT

We gratefully acknowledge financial support from Young teachers and personnel returning from overseas study research program Fund of Nankai University.

REFERENCES

[1] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In Proceedings of the Winter USENIX Technical Conference, 1994.

[2] Karl L. Swartz. 1997. Adding response time measurement of CIFS file server performance to NetBench. In Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997 (NT'97). USENIX Association, Berkeley, CA, USA, 12-12.

[3] FTP File System. http://ftpfs.sourceforge.net/

[4] Matthew E. Hoskins. 2006. SSHFS: super easy file access over SSH. Linux J. 2006, 146 (June 2006), 4-.

[5] Oleg Kiselyov. 1999. A network file system over HTTP: remote access and modification of files and files. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '99). USENIX Association, Berkeley, CA, USA, 31-31.

[6] S.Ghemawat, H.Gobioff, ST. Leung. The Google File System. In Proceedings of the 19th Symposium on Operating Systems Principles, Lake George, NY, USA, October, 2003.

[7] K.Shvachko, H.Kuang, S.Radia, R.Chansler. The Hadoop Distributed File System. In Proceedings of the 26th Symposium on Mass Storage Systems and Technologies, Incline Village, Nevada, USA, May, 2010.

[8] Dropbox. https://www.dropbox.com/about

[9] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and performance in a distributed file system. ACM Trans. Comput. Syst. 6, 1 (February 1988), 51-81. DOI=10.1145/35037.35059 http://doi.acm.org/10.1145/35037.35059

[10] Satyanarayanan, M.; Kistler, J.J.; Kumar, P.; Okasaki, M.E.; Siegel, E.H.; Steere, D.C.; , "Coda: a highly available file system for a distributed workstation environment," Computers, IEEE Transactions on , vol.39, no.4, pp.447-459, Apr 1990 doi: 10.1109/12.54838

[11] James J. Kistler and M. Satyanarayanan. 1992. Disconnected operation in the Coda File System. ACM Trans. Comput. Syst. 10, 1 (February 1992), 3-25. DOI=10.1145/146941.146942 http://doi.acm.org/10.1145/146941.146942

[12] Puneet Kumar and M. Satyanarayanan. 1995. Flexible and safe resolution of file conflicts. In Proceedings of the USENIX 1995 Technical Conference Proceedings (TCON'95). USENIX Association, Berkeley, CA, USA, 8-8.

[13] Mummert, L. B., Ebling, M. R., & Satyanarayanan, M (1995). Exploiting weak connectivity for mobile file access. Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP '95, 143-155. doi:10.1145/224056.224068

[14] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and performance in a distributed file system. ACM Trans. Comput. Syst. 6, 1 (February 1988), 51-81. DOI=10.1145/35037.35059 http://doi.acm.org/10.1145/35037.35059

[15] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. 2012. BlueSky: a cloud-backed file system for the enterprise. In Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST'12). USENIX Association, Berkeley, CA, USA, 19-19.

[16] Muthitacharoen, A., Chen, B., & Mazières, D (2001). A low-bandwidth network file system. Proceedings of the eighteenth ACM symposium on Operating systems principles - SOSP '01, 174-14. doi:10.1145/502034.502052

[17] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. 2007. SafeStore: a durable and practical storage system. In 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07), Jeff Chase and Srinivasan Seshan (Eds.). USENIX Association, Berkeley, CA, USA,Article 10 , 14 pages.

[18] Nelson, M. N., Welch, B. B., & Ousterhout, J. K (nd). Caching in the Sprite network file system. ACM Transactions on Computer Systems, 6(1), 134-154. doi:10.1145/35037.42183

[19] L. Hejtmánek, L. Matyska. Nonblocking Distributed Replication of Versioned Files. Journal of Software, Vol 2, No 5 (2007), 16-23, Nov 2007. doi:10.4304/jsw.2.5.16-23
[20] Stephen Smaldone, Vinod Ganapathy, and Liviu lftode. Working set-based access control for network file systems. In SACMAT'09: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, Stresa, Italy, June 2009.
[21] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems (TOCS), 10(1):26–52,1992.
[22] Ju, J., Wu, J., Fu, J., Lin, Z., & Zhang, J. (2011). A Survey on Cloud Storage. Journal Of Computers, 6(8), 1764-1771. doi:10.4304/jcp.6.8.1764-1771

Zhang-Chunxia received her Ph.D in The Chinese Academy of Sciences in 2005. She is working as at College of Electronic Information and Automation, Tianjin Science & Technology University in China. Her research interests include sensor network and distributed systems.

Shi-Baoxu is an undergraduate student of Software College of the Nankai University. His research interests include storage management and network.

Li-Xudong is currently working as an associate professor at the Nankai University, His research interests include software architecture and operating system. Li-Xudong is the corresponding author of this paper.