University of West Bohemia
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Distributed File Systems

The State of the Art and concept of Ph.D. Thesis

Pavel Bžoch

Technical Report No. DCSE/TR-2012-02
June, 2012

Distribution: public

Technical Report No. DCSE/TR-2012-02
June 2012

Distributed File Systems

Pavel Bžoch

Abstract

Need of storing a huge amount of data has grown over the past years. Whether data are of multimedia types (e.g. images, audio, or video) or are produced by scientific computation, they should be stored for future reuse or for sharing among users. Users also need their data as quick as possible. Data files can be stored on a local file system or on a distributed file system. Local file system provides the data quickly but does not have enough capacity for storing a huge amount of the data. On the other hand, a distributed file system provides many advantages such as reliability, scalability, security, capacity, etc.

In the report, we will provide the state of the art in DFS oriented on reliability and performance in these systems. First of all, traditional DFS like AFS, NFS and SMB will be explored. These DFS were chosen because of their frequent usage. Next, new trends in these systems with a focus on reliability and increasing performance will be discussed. These include the organization of data and metadata storage, usage of caching, and design of replication algorithms and algorithms for achieving reliability.

This work was supported by:

- *UWB* grant *SGS*-2010-028 Advanced Computer and Information Systems.

Copies of this report are available on http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Table of Content

List of Figures

List of Abbreviations

DS	Distributed System
FS	File System
DFS	Distributed File System
LAN	Local Area Network
MAN	Metropolitan Area Network
WAN	Wide Area Network
AFS	Andrew File System
NFS	Network File System
SMB	Server Message Block
IBM	International Business Machines Corporation
RPC	Remote Procedure Call
CIFS	Common Internet File System
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
ISO	International Organization for Standardization
OSI	Open Systems Interconnection
HDD	Hard disk drive
RAID	Redundant Array of Inexpensive/Independent Disks
EXT	Extended File system
XFS	Extended File System
NTFS	New Technology File System
HFS	Hierarchical File System
UPS	Uninterruptible Power Supply
DFSR	Distributed File System Replication
RDC	Remote Differential Compression
RAM	Random-Access Memory
CBR	Criteria Based Replication
CBRP	Criteria Based Replica Placement
CBPA	Criteria Based Primary-copy Assignment
P2P	Peer to Peer
DHT	Distributed Hash Table
ID	Identification/Identity/Identifier
IA	Interface Agent

WA	Working Agent
DMA	Domain Manage Agent
MMA	Main Management Agent
DCS	Domain Consistence Servers
OS	Operating System
LSM	Log-Structured Merge tree
FIFO	First In, First Out
LRU	Last Recently Used
LFU	Least Frequently Used
OPT	Optimal
GPRS	General Packet Radio Service
EDGE	Enhanced Data for GSM Evolution
HSDPA	High-Speed Downlink Packet Access
Wi-Fi	Wireless Fidelity

1 Introduction

1.1 Distributed system

Modern computations require powerful hardware. One way of gaining results faster is getting new hardware over and over again. Buying a supercomputer is, however, not a cheap solution. It also takes plenty of time to install software to these new supercomputers. Another way in achieving better system performance is using a distributed system. In a distributed system, several computers are connected together usually by LAN. Now, we can characterize distributed system with a simple definition:

> *A distributed system is a collection of independent computers (nodes) that appears to its users as a single coherent system.* [1]

This concept brings many advantages. Better performance can be achieved by adding new computers to the existing system. If any of the computers crashes, the system is still available. Using DS brings several problems too. In the distributed systems, we have to solve synchronization between computers, data consistency, fault tolerance etc. There are many algorithms which solve these problems. Some of them are described in [1].

1.2 Distributed file systems

Distributed file systems (DFS) are a part of distributed systems. DFS do not directly serve to data processing. They allow users to store and share data. They also allow users to work with these data as simply as if the data were stored on the user's own computer.

Compared to a traditional client-server solution, where the data are stored on one server, important or frequently required data in DFS can be stored on several nodes (node means a computer operating in a DFS). This is called *replication*. Replication can be used for achieving better system performance and/or for achieving *reliability* of the system.

The data in a DFS are more protected from a node failure. If one or more nodes fail, other nodes are able to provide all functionality. This property is also known as *availability* or *reliability*. The difference between availability and reliability is simple. Availability means that the system can serve client a request at a moment when the client connects to the system. Reliability means that the system is available all the time when the client is connected to it.

Files can also be moved among nodes. This is typically invoked by an administrator and it is done for improving a load-balancing among nodes. The

users should be unaware of where the services are located and also the transferring from a local machine to a remote one should also be transparent [2]. In DFS, this property is known as *transparency*.

If the capacity of the nodes is not enough for storing files, new nodes can be added to the existing DFS to increase DFS capacity. This property is known as *scalability*.

A client usually communicates with the DFS using LAN, which is not a secure environment. Clients must prove their identity, which can be done by authenticating themselves to an authentication entity in the system. The data which flow between the client and the node must be resistant against attackers. This property is known as *security*.

Next chapters describe how are these properties solved in traditional distributed file systems (chapter 3) and modern trends in distributed file systems (chapter 4). First of all, traditional distributed file systems will be described (chapter 2).

Chapter 5 will describe our future fork. In this chapter, we will focus on mobile devices, and on algorithms which can be used for achieving performance and reliability on these devices. Accessing files from mobile devices requires algorithms which take into account changing communication channels caused by user's movement.

2 Traditional distributed file systems

In this chapter, we will provide overview of traditional distributed file systems. These systems are called traditional because of their frequent usage. Also, many new solutions are based on these systems. Some of the traditional DFS are commercial (like AFS), and others are free (OpenAFS, NFS, Coda and Samba). In this chapter, we will focus on NFS4, OpenAFS, Coda and SMB.

Before we describe traditional DFS, we will focus on DFS generally. Every distributed file system consists of several nodes (a node is a computer operating in DFS). Some of these nodes serve for storing file content; others serve for storing metadata about files. File content is usually stored on a local file system. Metadata are usually stored in database. Every database record must also have a link to the file storage to the file content.

File content and metadata are usually created during file upload process. A user send whole file with its attributes to DFS. On the server side, file content and metadata are separated and sent to relevant storage. Whole upload process is depicted in Figure 1.

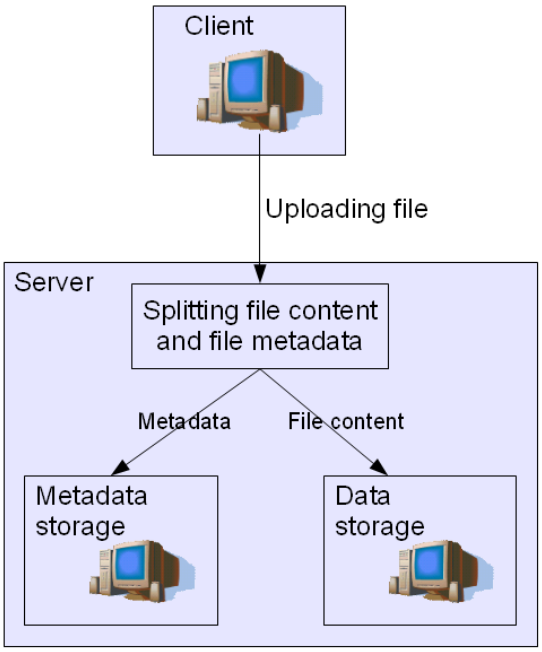

Fig. 1 File uploads process

When clients want to download the file back to their computers, they contact DFS. In DFS, the metadata storage provides attributes of the file and a link to the data storage where the file content is stored. This information is then sent back to the client. The client then contacts the data storage for getting the file content. The

whole download process is depicted in Figure 2. In this case, we don't take into consideration file locking.

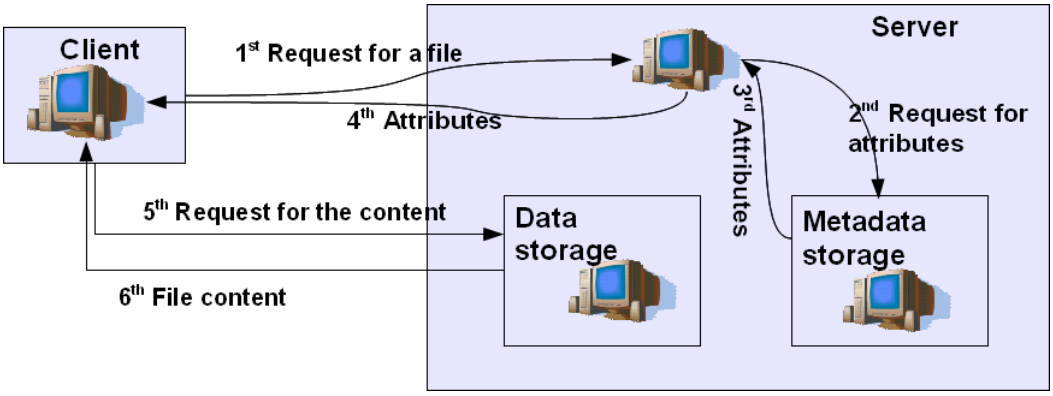

Fig. 2 File downloads process

2.1 AFS

AFS (Andrew File System) was originally created at Carnegie Mellon University; later it became a commercial product supported by IBM. Now it is being developed under a public license (OpenAFS). The design goal of AFS was to create a system for large networks [3].

The smallest operating entity in AFS is *whole file* which is the basic unit of data movement and storage. Whole file was chosen rather than some smaller unit such as physical or logical record. [4]

AFS uses a uniform directory structure on every node. The root directory is /afs. This directory contains other directories which correspond to the *cells*. Cells usually represent several servers which are administratively and logically connected. One cell consists of one or more *volumes*. One volume represents a directory sub-structure, which usually belongs to one user. These volumes can be located on any AFS server. Volumes can be also moved from one AFS server to another. Moving volumes does not influence the directory structure.

Information about the whole system is stored in a special database server [5]. For minimizing client-server communication, AFS supports client-side caching. Cached files can be stored on a local hard disk or in a local memory. Frequently used files are permanently stored in this cache.

AFS does not provide access rights for each file stored in the system, but it provides directory rights. Each file inherits access rights from the directory where the file is located. An access list specifies various users (or groups of users) and, for each of them, specifies the class of operations they may perform [4].

For achieving better performance of read-only files, *snapshots* or *clones* are used. These snapshots are then stored on *replica servers*. Snapshots are usually made

periodically. When the other servers are overloaded, the replica server provides files to the clients instead of these servers.

AFS uses Kerberos [6] as an authentication and authorization mechanism. More information about AFS can be found in [7]. AFS is a very stable and robust system and it is often used at universities.

2.2 NFS

NFS (Network File System) is an internet protocol which was originally created by Sun Microsystems in 1985, and was designed for mounting disk partitions located on remote computers. NFS is based on RPC (Remote Procedure Call) and is supported in almost all operating systems. The NFS client and server are parts of the Linux kernel. The Kerberos system is used for user authentication. NFS was made for making client unaware of location of their files. NFS remote file system can be mounted into local directory structure. Clients can then work with their files as if the files were stored on their local file system. On server side, NFS uses the same directory structure as is shown after mounting to the client. Currently, NFS exists in several versions. In the next text, we will focus on the latest version NFSv4.

In NFS, there usually exists an *automounter* on client side [8]. An automounter is a daemon which automatically mounts and unmounts NFS file system as needed. It also provides ability to mount another file partition if the primary partition is not available at a given moment. List of replicas must be made before automounter daemon is run.

In NFSv4, system performance is increased by using a local client cache. NFS can be extended into pNFS (parallel NFS), which contains one more server called metadata server. The metadata server can connect a file system from any data server to a virtual file system. It also provides information about the file location to the clients. When clients write file content, they must also ensure file updating on all servers where the file is located.

NFS communicates on one port since version 4 (previous versions used more ports), so it is easy to set up a firewall for using NFS4 [5]. The difficulty with NFSv4 is with clients. There do not exist suitable clients for all operating systems. More information about NFS can be found in [9].

2.3 Coda

Coda was developed at Carnegie Mellon University in 1990. It is based on the AFS idea and is implemented as a client and several servers. This system was mainly designed to achieve high availability.

The client uses a local cache. Cached files can be used by clients even after disconnection from Coda server. This feature is called *disconnected operation*. While the client is disconnected from the server, all changes made to files are stored in a local cache. After reconnecting to the server, all these changes are propagated to the server. If any collision occurs, the user has to solve it manually.

Coda uses Kerberos as an authentication and authorization mechanism. Servers provide file replication for achieving availability and safety. Coda uses RPC2 for communication. Servers store information about files which are in the client's cache [3]. When one of the cached files is updated, the server marks this file as non-valid.

The difference between Coda and AFS is in replication. Both of these systems use replication for achieving reliability. Coda uses optimistic replication; AFS uses pessimistic replication method. Pessimistic replication means that the replicas in AFS are read-only and present a snapshot of the system. Optimistic replication means that all replicas are writable. Client in Coda system must ensure file updating in all given replicas.

2.4 SMB (Samba)

All mentioned distributed file systems were originally made for Linux/UNIX systems. SMB was developed in 1985 by IBM as a protocol for sharing files and printers. In 1998 Microsoft developed a new version of SMB called Common Internet File System (CIFS), which uses TCP/IP for communication.

SMB has been ported to other operating systems where the SMB is called *samba*. This system is stable, wide-spread and comfortable. SMB can use Kerberos for authentication and authorization of users. It does not use local client-side caching. SMB uses the operating system's file access rights. SMB is wide used in Windows™ operating systems.

2.5 Summary

All mentioned traditional DFS were developed some time ago. They use algorithms that were commonly used at this time. Since the development of these DFSs was finished, new algorithms for increasing reliability and performance were evolved. In my opinion, there still exist challenges to improve these algorithms or to develop new algorithms.

3 Reliability and performance in traditional DFS

In this chapter, we will focus on reliability and performance. Reliability is described in section 3.1, and increasing performance is described in section 3.2.

3.1 Reliability in traditional DFS

Recall to the Introduction, reliability is a property which guarantees clients all functionality all the time when clients are connected to the system. Reliability in DFS can be achieved by using several techniques and methods. We can divide these methods into three categories: reliable communication, one node reliability, and reliability of the whole system.

3.1.1 Reliable communication

For communication between the DFS and a client, a computer network is usually used. In a computer network, several protocols can be used. These protocols can be connectionless or connection-oriented. In the mostly used TCP/IP model, UDP is a connectionless protocol, and TCP is a connection-oriented protocol. For achieving reliability, DFS usually use connection-oriented protocols. Both TCP and UDP protocols are protocols on transport layer in ISO/OSI model as is depicted in Fig. 3. Each of these protocols uses port for different application which the message will be delivered to. TCP and UDP provide end-to-end connection.

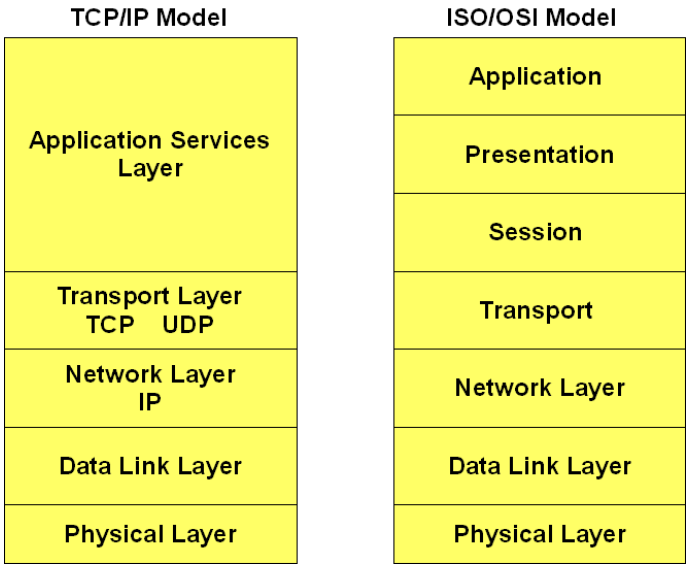

Fig. 3 Difference between TCP/IP model and ISO/OSI model

Advantage of this solution is that the client knows almost immediately that a failure occurred and can attempt to set up a new connection. Disadvantage of using connection-oriented protocols is slowness of these protocols in comparison to connectionless protocols. Connection-oriented protocols usually have bigger overhead than connectionless protocols.

3.1.2 One node reliability

Once the communication is solved by using connection-oriented protocol, reliability is the next issue to be solved on the server side. Reliability on the server side in DFS can be solved by using several techniques.

Reliability of one node means that the node is still service-able while there is a fault (e.g. HDD is down). HDD crash can be prevented by using RAID. RAID is an acronym for Redundant Array of Inexpensive/Independent Disks. While using RAID, we prevent a failure by using more hard disk. There exists several methods in storing data on RAID and there exists some kind of RAID's:

- RAID 1 (HDD mirroring). In the system, there must be two hard disks. Same data are then stored on both of these disks. This option guarantees that the node provides all functionality when one HDD fails.
- RAID 2 (bit-level striping with dedicated Hamming-code parity). This RAID requires (*N+1*) hard disks. The data are stripped by bites over *N* disks. On the last disk, parity Hamming-code is stored. This option guarantees that the RAID is protected from one hard disk failure. If data disk crashes, the data can be recovered from others and from Hamming-code. If the last disk fails, Hamming-code can be calculated from others hard disks. Parity disk is a bottle-neck of the system. If there is write request to the system, parity must be recalculated and restored.
- RAID 3 (byte-level striping with dedicated parity). RAID 3 is similar to RAID 2 but it uses bytes striping instead of bit striping.
- RAID 4 (block-level striping with dedicated parity). The data are stripped by blocks. Parity is stored on dedicated disk. Otherwise, RAID 4 is similar to RAID 2 and 3.
- RAID 5 (block-level striping with distributed parity). The data are stripped by blocks and stored on hard disks. Parity is distributed over all disks in RAID5. Distributed parity eliminates bottle neck from RAID 2-4 which have dedicated parity disk. RAID 5 needs at least three hard disks, and it is resistant to one disk failure.
- RAID 6 (block-level striping with double distributed parity). RAID 6 is similar to RAID 5, but it uses two methods in calculating parity. Parity is distributed over participated disks. RAID 6 requires at least four disks, and is protected from two disks failure.

Not all of named methods are suitable for using for storing data. RAID 1, RAID 5 and RAID 6 are commonly used. RAID 1 is faster than RAID 5 and 6, but it has lover capacity compared to RAID 5 and 6. RAID 5 and 6 has bigger capacity, but we must re-/calculate parity while writing file content to these systems.

Preventing inconsistent state of the file system can be also done by choosing a suitable file system which will be used for storing data. File systems use journal techniques to prevent an inconsistent state. A journal technique or strategy describes when and for what will be the journal used for.

Mostly used strategies are:

- *Writeback mode.* While using this strategy, data blocks are directly written to the disks. Metadata is journaled. This approach prevents metadata corruption, but data corruption can occur (if metadata is updated before the data is written to the disk and the system crashes). [10]
- *Ordered mode.* To prevent an inconsistent state in the writeback mode, ordered mode writes the data block before journaling the metadata. If the system crashes during writing data, the system can simply turn back to a consistent state. [10]
- *Data mode.* In this mode, both metadata and data are journaled before writing changes to the disk. This degree of protection provides the highest level of disk protection against corruption. On the other hand, this strategy is the slowest, because it must write data twice. Once the data are written to the journal and then to the disk. [10]

A journal technique has usually four steps:

1) The first step is writing a change in the file system into the journal.
2) The second step is applying the change to the file system.
3) The third step is writing the end of the operation into the journal.
4) The fourth step is clearing the record from the journal.

Journal techniques use e.g. EXT3, EXT4, ReiserFS or XFS in Linux/Unix systems, NTFS in Microsoft Windows systems, HFS+ in Mac OS X systems. AFS and Coda require a journal file system. Additionally, AFS client needs EXT2 file system for storing a local cache [11]. NFS can be run on both journal and non-journal file systems.

File system can be also damaged when the power supply is down. Preventing this state can be done by using an Uninterruptible Power Supply (UPS). UPS allows the nodes to save all critical data to prevent file system inconsistency in case of power shortage.

3.1.3 Whole system reliability

Reliability of the whole system can be achieved by replicating files. Replication in DFS means that the files are stored on several nodes in DFS. When one of the nodes crashes, the file is still available from the other node.

Replication can be done automatically or administratively. Both, AFS and NFS, use administrative replication. Administrative file replication means that the administrator chooses what will be replicated, and a location for replicas. None of the traditional DFS uses automatically replication.

In AFS and Coda, the smallest replication entity is volume [12]. NFS supports file replication since version *4*, previous version did not support replication. NFS can replicate only the whole file system [1].

File replication can be done in an optimistic and a pessimistic way. A pessimistic way means that the client can write only to a master replica, other replicas are read-only and are maintained administratively. An optimistic way means that all replicas are writable. In this case, while data is being updated, the client has to update all replicas. Another option is that client updates one replica and DFS propagates these changes to all replicas. AFS and NFS use pessimistic way [**7**]. Coda uses optimistic way [12]. During replication, all mentioned DFS also support file locking. There are usually two types of locks, an exclusive lock for writing and a shared lock for reading file content.

3.2 Increasing performance

Performance in traditional DFS can be increased by using a file replication and a caching mechanism. Replication is usually used for increasing server performance. Caching algorithms can be used on both side of the communication.

3.2.1 Replication

In this sub-section, we will focus on replication for achieving performance. When we replicate the files, the original file is usually called *the primary replica* or *master replica;* other copies are called *replicas*.

By using replication, choosing the file and the place for replication is very important [13]. The file for replication should be read very often and should not be modified very often. Writing or updating a replicated file is an expensive operation. Choosing a place for a file replica is also very important. The server which is chosen to store the replica should not be over-loaded and should have good network connectivity.

AFS, CODA and NFS use administrative replication. AFS and NFS use pessimistic file replication. Coda uses optimistic file replication.

For using in Windows™, Microsoft developed a Distributed File System Replication (DFSR) service. This service provides *multi-master* replication and keeps folders synchronized on multiple servers [14]. DFSR uses a new compression algorithm called Remote Differential Compression (RDC). RDC can

be used to efficiently update files over a limited-bandwidth network. RDC detects removals, insertions, and rearrangements of data in files. Based on this information, DFSR replicates only the deltas (changes) when files are updated [14].

3.2.2 Caching

A *cache* in the computer system is a component which stores data that were frequently requested, hence can be potentially used in the future. When the cached data are requested, the response time is shorter than when the data are not in a cache and must be downloaded. The cache can be stored in RAM for fast access and/or on hard disk.

A cache can be on both side of communication. On server side, the cache is usually located in RAM. On client side, the cache can be located in RAM or on hard disk. On server side, if file content is cached, cache mechanism spares time because there is no need to access file content from hard disk. On client side, if the client requests file, cache mechanism spares time because there is no need to communicate with a server.

Client-side caching is also sometimes called client initiated replication. Client cache can also provide so-called *offline cache.* This means that the client can access files from cache after disconnection from the server. Offline cache is often stored on hard disk. Off-line caching mechanism is used in Coda.

4 New trends in distributed file systems

In this chapter, we will provide information about state of the art, and new trends in distributed file systems. We will explore algorithms that provide reliability and increase performance in DFS. We will start with algorithms which are used for achieving reliability.

4.1 Reliability

For achieving reliability in distributed file systems, the first step is choosing a suitable file system for storing data and/or metadata. As was mentioned in chapter 3.1.2, reliable file systems usually use journaling. The journal technique can be also used for keeping whole DFS consistency. The second step in increasing reliability is using file replication. File replication can be also used in achieving better system performance. We will discuss the difference between replication for performance and replication for reliability in chapter 4.2.3.

4.1.1 Journaling

A journal in DFS can be used as a write-ahead commit log for changes to the file system that must remain unchanged [15]. Every transaction made by a client is recorded in the journal, and the journal file is flushed and synchronized before the change is committed back to the client [15].

Paper [15] presents another use of the journal, as a checkpoint. The checkpoint is a summarization of all records that were written to the journal, and presents a consistent state of the DFS. When any fault appears, the DFS can recover by returning to this checkpoint by applying changes which are stored in the journal. Journal and checkpoint serve for maintaining metadata information. If journal records or/and checkpoint are missing or are corrupted, namespace information is lost (in this concept). To prevent these incidents, critical information about namespace is stored on several storages.

4.1.2 File replication

Recall that file replication in a traditional DFS is usually made administratively. Papers [16] and [17] present a new technique of file replication: dynamic file replication. Both papers use statistical information about files and servers. Based on these statistics, it can be decided which files will be replicated and where. A replication algorithm is depicted in Figure 4.

In the case which is depicted in Fig. 4, many remote users request the file A. If the number of requests reaches the threshold value, the replication process starts at the Node 1 (Primary replica holder). The file A is then sent to the new replica holder (Node 2). This node becomes new replica holder. Simultaneously, the file A

is sent to Node 3, which becomes new replica holder too. After replication process ends, all three nodes can provide file A.

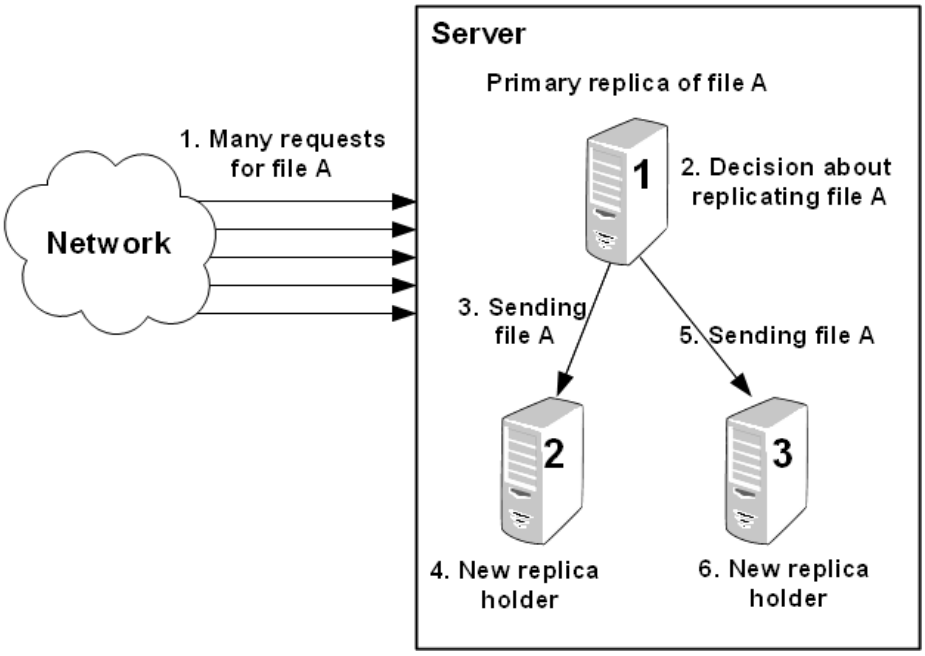

Fig. 4 File replication

Paper [16] presents the Criteria Based Replication (CBR). This model uses two main algorithms for achieving high availability and reliability. The first algorithm is the Criteria Based Replica Placement algorithm (CBRP). This algorithm collects information about physical location of the server, load of the server, etc. The second algorithm is the Criteria Based Primary-copy Assignment (CBPA). This algorithm is used for choosing the primary copy (or replica). This is used for making primary copy available for maximum number of clients at any single session. Statistics for these two algorithms are collected through system calls and by predefined system variables.

The Criteria Based Replica Placement algorithm monitors each criterion individually [16]. Then, it periodically calculates the result for each criterion. If this result exceeds a threshold value, the file is replicated.

In the Criteria Based Primary-copy Assignment algorithm, the server is being chosen for holding the primary replica of a file. This algorithm makes a list of servers with chronological priority to be a primary-copy server [16]. The decision of choosing primary copy server shall be done before client requests for a file. This file is then highly available for the client.

Paper [17] presents storing information about a whole system such as the service ratio of peers, reliable value of peers, etc. Based on this information, the system can place a replica at the most reliable place at a particular time. The whole system

in this conception is divided into *peers* and *super peers*. Super peer is a computer which is rich in resource and capability, and is used to manage peers in its group [17].

Super peer also collects statistical information about peers in group and runs replica management service. This service has fully knowledge about master replica location, network topology and bandwidths to the relevant peers [17]. The decision for making the new replica and the new replica placement is made by the super peer. The super peers maintain a list of frequently requested files, and also collect information about average response time. This list is periodically updated. If the response time for any file exceeds threshold value, the file is replicated.

The previous two papers [16] and [17] present methods for file replication. These two papers do not mention algorithms which can be used for updating files and their replicas. Paper [18] uses file replication for achieving reliability too. In addition to previous two papers, it presents the master-slave full replication method. This method updates the master replica first; other replicas are updated only when there is a need to receive the data from them. For achieving data integrity, this DFS uses content hash. Master-slave replication is used for increasing performance and decreasing response time.

Another way of storing file content is presented in paper [19]. This paper presents a DFS based on a P2P network. An example of file storage is depicted in Fig. 5.

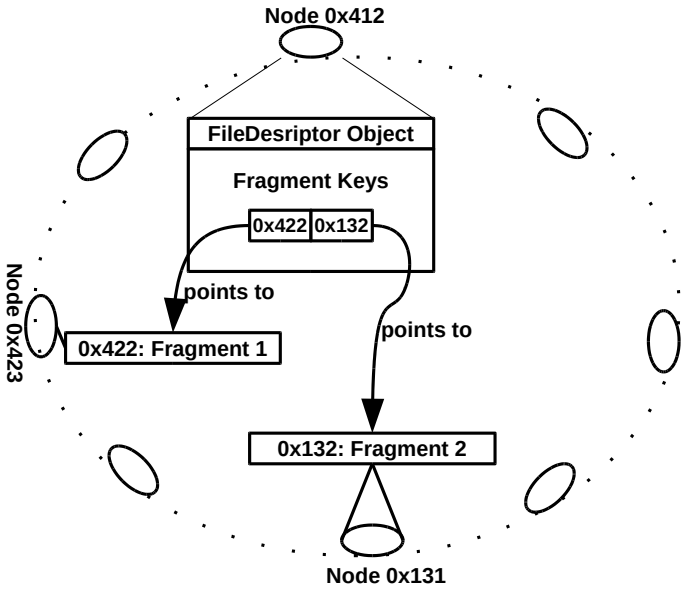

Fig. 5 Example of file storage in the P2P overlay [19]

Files in this system are split into fragments, which are then stored on clients (peers). Links to these fragments are stored in a Distributed Hash Table (DHT). The DTH also maintains file and directory descriptors. Every client in this system

is responsible for files, whose fragment keys are near the peer ID. A fragment key is made from a 160-bit random value and a combination of a path name and a fragment number as a domain [19]. File descriptor is responsible for storing the fragment keys.

Advantage of using path name calculating hash is in updating file content. If new data is written to the fragments, there is no need to update DHT records. Fragments are still on the same nodes [19]. DHT records must be updated only when the file is moved between directories. In this circumstance, file fragments must be moved to new nodes according to the new calculated fragment keys.

This DFS uses two ways in file replication: passive and active. Passive file replication is done by using file caching. In the cache, frequently asked fragments are temporarily stored. Active file replication works as follows. The peer, which ID is closest to the fragment ID, is responsible for a fragment replication [19]. This peer checks periodically if there are enough replicas in the system. If any of the peer checks its status to *leave*, the others peers must check if they are now responsible for the fragment.

File replication for achieving reliability is also used in other DFS. CloudStore [20] typically uses 3-way file replication, but administrator can set up to 64 replicas of one file. If there is a need for replication (e.g. node outage), a metadata server can replicate a file chunk to another node. This conception of file replication for achieving reliability is derived from the Google File System [21].

Reliability in GlusterFS [22] uses three file distribution methods. The first one is Distribute-only. In this conception, each file is stored only once. This solution does not provide increased reliability. Reliability can be achieved by using the second method: distribution over mirrors. This means that each storage server is replicated to another storage server. The third method is stripping large files over nodes in DFS. This is usually used for very large files (minimum is 50GB per file). The first and the third method are less reliable then the second method.

4.1.3 File locking

While achieving reliability in DFS, data consistency is also very important. Data consistency can be achieved by using file locking. There are usually two types of locks: a shared lock for file reading and an exclusive lock for file writing. A shared lock is used when the data can be read simultaneously by many clients, but cannot be written simultaneously. While a file is being read by clients, other clients cannot write into this file. An exclusive lock means that only one client can access a locked file and can write file content.

Paper [23] presents a DFS based on mobile agents. The mobile agents are used for communication, synchronization, for data access, and for maintaining consistency by using two layer lock request mechanism. In this system, there exist four kinds of agents:

- Interface Agent (IA) accepts and process file systems calls from client, and coordinates with others agents.
- Working Agent (WA) accept calls from IA, and executes file operation on remove server (WA moves itself to this server).
- Domain Manage Agent (DMA) is responsible for cache, name, space management, access control management in the domain [23].
- Main Management Agent (MMA) is responsible for coordination DMAs.

There is one Main Consistence Server (MCS) with one MMA and many Domain Consistence Servers (DCS) with theirs DMAs in this conception. For the file locking, there is a two-layer lock request mechanism. The two-layer lock system is depicted in Fig. 6. Obtaining a file lock is done by asking DCS for it (IA→WA→DMA), if DMA does not have the requested lock, then the lock must be obtained at MMA (DMA→MMA).

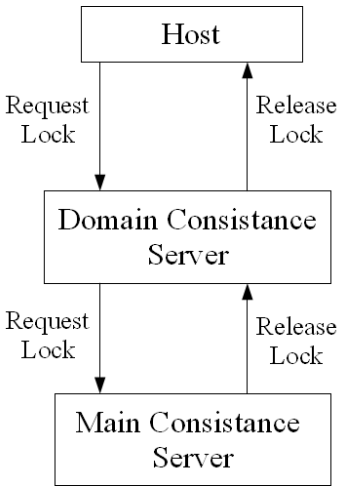

Fig. 6 The two layer lock request mechanism [23]

4.2 Increasing performance

This section will describe modern trends in DFS with a focus on increasing performance. First of all we will describe trends in data storage and metadata storage organisation. Data storage is used for storing file content. Metadata storage usually stores file attributes and links to the file content in the data storage. Afterwards, we will focus on algorithms which are commonly used for increasing performance: replication and caching algorithms.

4.2.1 Data Storage

Data storage is used for storing file content. When users want to upload a file to the DFS, they send the entire file to the server. On the server side, this file is split into two parts: file content and file metadata. File content is then stored in a data storage node. Uploading a file to a server and storing file content is depicted in Fig. 7.

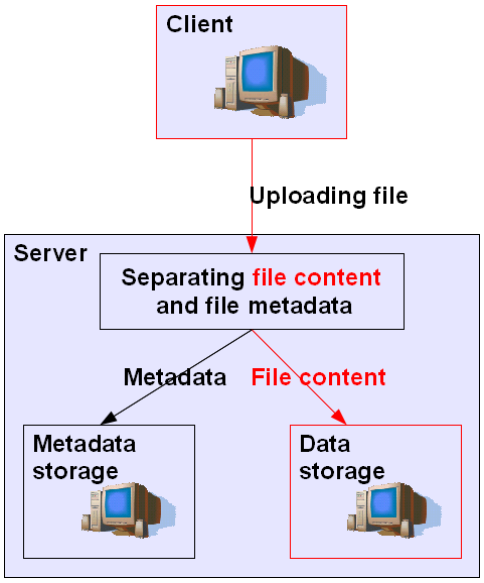

Fig. 7 File uploading process – file content

For the data storage, local hard disks with their own file systems are usually used. On nodes with OS Microsoft Windows, NTFS is commonly used. In UNIX-like systems, several different file systems exist. Not all of these systems are suitable for all types of files.

According to the tests in [24] the ReiserFS is more efficient in storing and accessing small files, but it has a long mount time and is less reliable than EXT2/3. XFS and JFS have good throughput, but they are not efficient in file creation. EXT2/3 has severe file fragmentation, degrading performance significantly in an aged file system [24]. The decision on which file system will be used for data storage is an important part of DFS design.

Another method of storing file content is a designing new data organization of a hard disk. This concept is used when existing data organization (file system) of a hard disk is not suitable for files which will be stored there. New hard disk organization [**25**] is depicted in Fig. 8.

In this proposal, the superblock is located at the beginning of the file system. Next to the superblock, the disk block bitmap is situated. There is no need for the inode section, because *inodes* are spread over the entire disk. The disk block bitmap serves for recording weather a block is used or not [**25**]. Every single inode can be

locked without increasing the total amount of locks. Distribution of the inodes in this proposal also makes the system more expandable because the distribution makes it possible for the number of inodes to increase or decrease dynamically [**25**].

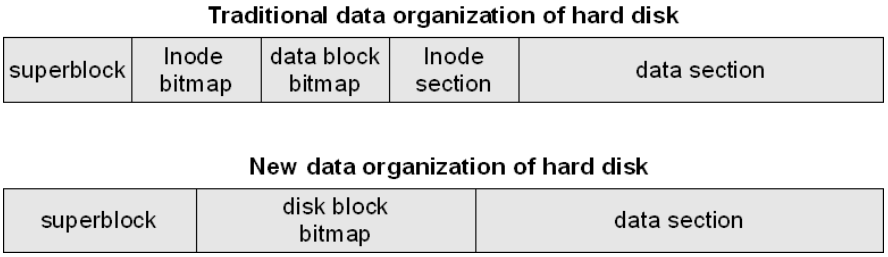

Fig. 8 Data organization of hard disk [25]

Writing and reading file content or creating a new file is a slow operation. I/O operations are the bottleneck in achieving better performance in DFS. Uploading and storing files on a server has several steps. These steps must be done chronologically to ensure data consistency. The steps are: sending a file to the server → splitting file content and file metadata → creating a new metadata record → creating a new file and storing file content → setting file attributes → connecting metadata with the file handle. Both, creating the metadata record and storing content, are slow operations. According to [26] and [**27**], these slow operations can be accelerated.

Paper [26] presents increasing performance by making changes in an upload protocol. These changes can be made in different ways:

- *Compound operations*. In this proposal, we suppose that the steps in upload protocol are independent from the others. So we can do these steps in parallel. E.g. we can create a new metadata record and set attributes in one step. This reduces the amount of sent messages during upload process.
- *Pre-creation of data files* at the data storage servers. Creating a new file and getting a file handle for connecting with a metadata record is a slow operation. We can create file handles before the file is uploaded to the server and then we can upload the file and make the metadata record parallel.
- *Leased handles*. In this proposal, a client has leased IO handles (from a data server). If the client wants to upload a file to the server, the client application can use one of the leased IO handles. Creation metadata record and file uploading can be done in parallel.

Paper [27] presents increasing performance by using methods which presume uploading huge amount of small files. Method pre-creating file object is similar to [26]. Other methods are:

- *Stuffing*. While using this method, the first block of a new created file is stuffed with stuffing bits. This concept supposes that the uploaded files will be small. Client application can create a metadata record in parallel with uploading file content and if the file is small, there is no need to allocate more blocks on the hard disk.
- *Coalescing Metadata Commits*. If the client application uploads many small files, creating and storing metadata records takes long time. At the metadata storage, we can collect new metadata records and flush them into database periodically or after reaching threshold value. This proposal decreases time which is necessary for creating metadata records.
- *Eager I/O*. While uploading a file to the server, we usually send two messages. The first message is for creating a file handler (answer to this message is file handler), the second message is a file content. If we presume that we always get file handler, we can merge these two messages into one. This proposal saves one message.

Both [26] and [27] demand cooperation between the file storage nodes and the metadata storage nodes.

Papers [24], [25], [26] and [27] assume that the whole file is stored in one node. Another way of storing files is splitting a file content into file fragments and storing these fragments on the client side. This proposal does not work on a client-server model, but works in P2P networks. Links to the file fragments are stored in a distributed hash table. The entire system is described in chapter 4.1.2. The P2P system architecture is depicted in Figure 5.

4.2.2 Metadata storage

Metadata is a specific type of data which gives us information about a certain item's content. In DFS, metadata is used for providing information about files which are stored in the data storage. This information is usually called file attributes. These attributes are the date and time of file creation, the date and time of the last modification, the file size, the file owner, the file access rights, etc.

Metadata storage also provides information about a directory structure. All this information is created during the upload process (see Figure 9). Each record must also have a link to the data storage. Metadata storage must provide functions for getting and storing file metadata, file searching, moving files within directories, deleting files and creating files. Additionally, metadata storage can provide locks for ensuring consistency during the file access. There are usually two types of locks: a shared lock for file reading and an exclusive lock for writing or updating file content. Metadata is usually stored in a database or in tree.

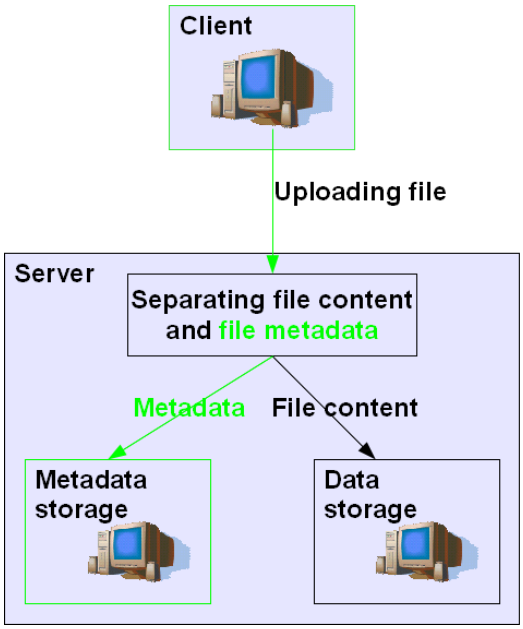

Fig. 9 File uploading process - metadata

Database records are used, e.g., in the AFS. While using the database, all metadata operations are represented by a database query. Trees are used e.g. in [28] and [15]. Entire tree is usually stored in RAM. Tree is used for maintaining namespace information. Adding a new file metadata record is simply adding a new node to the tree. This node must also have a link to the file content in the data storage. If the system uses file replication then the node corresponding to the file must have links to all replicas. Furthermore, the metadata server can provide load balancing while choosing suitable data storage for the clients.

Paper [28] presents DFS which uses one metadata server per cluster. This metadata server manages all file system metadata. Additionally, metadata server provides garbage collection for orphaned files, and provides services for file migration. Metadata server in this solution communicates with the data storage, gives the data storage information and collects statistical information about these servers. All metadata is stored in RAM (for increasing performance), and on hard disk (for increasing reliability). When the client requests a file, metadata server returns links to all data storages where file replicas are stored. Client stores this metadata in cache.

If the client in DFS [15] wants to read a file, the metadata server returns a link to the data storage which is closest to the client. The metadata node keeps a list of available data nodes by receiving heartbeat messages from these nodes. When a client wants to upload a file to the DFS, metadata server nominates three data storage for storing file content. The client must then arrange the file replication.

Another way of storing metadata involves using a log-structured merge tree (LSM). LSM tree is multi-version data structures composed of several in-memory

trees and an on-disk index [29]. Paper [29] describes database which consists of a set of indices, a log manager, and a checkpointer. Indices are data structures which are optimized for searching and storing database records. The log manager is used for persistently logging database modifications. Database log can be used for restoring database when the system crashes.

In the database, an index consists of a list of *N* in-memory trees and a single on-disk index [29]. The changes to the metadata are inserted to the *active* tree (last tree). All others trees and on-disk index are read-only. While looking for a record, the system searches through all in-memory trees from *N* to *1*. If the record was not found, the system would look into on-disk index. This method provides latest version of a record. The example of the database is depicted in Fig. 10.

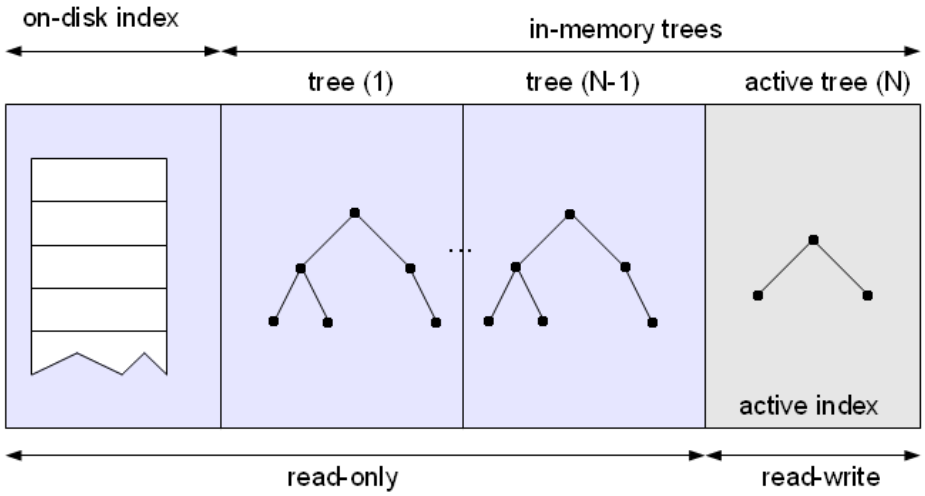

Fig. 10 Database with N in memory trees and on-disk index [29]

4.2.3 Replication

We have already mentioned replication algorithms for achieving reliability (chapter 4.1.2). In this chapter, we have said that replication for achieving reliability chooses place for replication based on reliability of the node. Now, we will briefly focus on replication for achieving better performance.

In this kind of replication, choosing the file and the place for replication is very important. The file for replication should be read very often and should not be modified very often. Writing or updating a replicated file is an expensive operation. Choosing a place for a file replica is also very important. The server which is chosen to store the replica should not be over-loaded and should have good network connectivity. These properties should be taken into consideration whether the file will be replicated or not.

4.2.4 Caching

Recall to the section 3.2.2, cache in the computer system is a component which stores data that can be potentially used in the future. The cache has limited capacity hence exist caching policies which try to predict future requests.

Caching policies need to mark the entity which can be removed from the cache when a new entity comes to the cache. Most of these algorithms are based on statistics made from previous data requests. These policies can be divided into basic policies and sophisticated policies. Basic policies don't use any statistics, sophisticated policies use statistics. Now, we will shortly describe algorithms used in these policies.

Basic policies are Random, and FIFO:

- The *random* replacement policy removes blocks for new files randomly. This policy is very easy to implement and is often used for comparison to others policies.
- The *FIFO* (First In, First Out) replacement policy maintains a queue of cached files. If a new file comes into the cache, a file is put at the tail of the queue. If the capacity of the cache is not enough for storing the new file, the file from the front of the queue is removed from the cache.

Sophisticated policies are FIFO with 2nd chance, LRU, LFU, and OPT:

- The *FIFO with 2nd chance* replacement strategy is similar to FIFO strategy. Additionally, it stores a flag for each file. This flag is used for marking files which have been read lately. If the file in the queue has this flag set, the caching strategy will only reset this flag and tries to remove next file in the queue. Only files with non-set flag can be removed from the cache.
- The *LRU* (Least Recently Used) replacement strategy stores for each file in the cache time when the file was accessed for the last time. The file which was accessed before the longest time is removed from the cache if new file comes to the cache. This concept assumes that the files which have been read recently will be read in the future again.
- The *LFU* (Least Frequently Used) replacement strategy stores for each file a number of accesses of the file. The file for taking out from the cache is the file with the lowest accesses count. While using this strategy, the strategy periodically reduces the count of hits because of ageing of the files in the cache. If the strategy does not reduce this count, the old files with huge count will be never removed from the cache.

- The *OPT* (Optimal) strategy chooses a file to be removed from the cache based on when it will be used again in the future [30]. The file that will be used farthest in the future will be removed first.

The most effective replacement policy is OPT, but OPT cannot be implemented in practice since that would require the ability to look into the future. According to [30], the most effective implementable replacement policy in DFS is LRU.

Many papers describe which of the cache policies is the most effective for use in a DFS. There are also some modifications of these policies for increasing cache-hit ratio. For caching large files, paper [31] extends existing LRU and LFU policies with a Size and Threshold policy. A LRU or LFU policy makes an ordered list of files which can be removed from the cache according to the LRU or LFU algorithm. LRU or LFU with Size means that the size of the file which will be removed must be greater or equal to the size of the new file. LRU or LFU with Threshold means that the size of the file which will be removed must be greater or equal to the threshold value. The most effective policy in [31] is, again, LRU with no extension.

Recall to the section 3.2.2, cache can be used on server side, on client side or on both side of the system. The server stores in the cache the data which are frequently requested by clients (see Figure 11). The client stores in the cache the data which may be requested again in the future (see Figure 12). If there are both caches in the system, the server cache-miss ratio increases. Clients often request files in their own cache, so they do not need the server to get the file. On the other hand, the server gets requests on different files, so the server-side cache is useless. This case is described in [32].

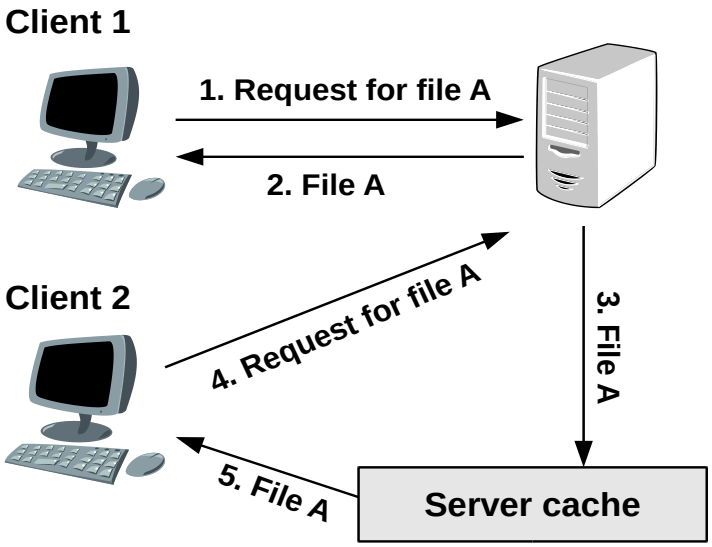

Fig. 11 Server-side caching

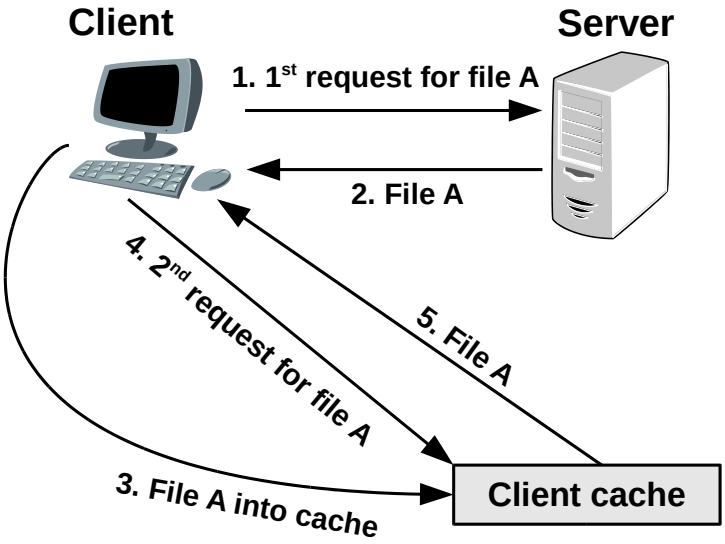

Fig. 12 Client-side caching

Paper [33] presents a decentralized collective caching architecture. In this concept, caches on the client side are shared among clients. When a client downloads a file, this file is then stored in the client's cache. The server stores a list of clients for each file. This list also contains the client network address. When another client wants to access this file, the server returns this list. The client can then download the file from one of the listed clients. The server then adds this new client to the list. This proposal decreases server work-load, but it requires cooperation among clients.

Collective caching architecture provides close-to-open consistency. Central server maintains commit timestamp (logical clock per every shared file). This number is increased every time a client commits new file content. When a client wants to download a file, a client application gets timestamp and a list with other clients holding requested file in their caches. Then the client looks into his own cache whether he has the file. If the file is found in the cache, the timestamp is verified. If the file is old, new content is downloaded either from other client (if any client has the file) or from the server.

Another way to reduce server workload and network bandwidth is by using proxy caching. Proxy caching in the DFS is introduced in [34]. A proxy cache stores files which are requested by clients. The architecture of proxy caching is depicted on Fig. 13. The whole cache is stored in a proxy server. The proxy server in this paper is on a local network. This paper also assumes that the connection to the server is slow (typically uses WAN). All file requests to the remote server pass through this proxy server. If the requested file is in the proxy cache, the proxy server returns this file and there is no need to communicate with the remote

server. Compared to the collective caching, proxy cache is a cache which is shared among users on local network segment.

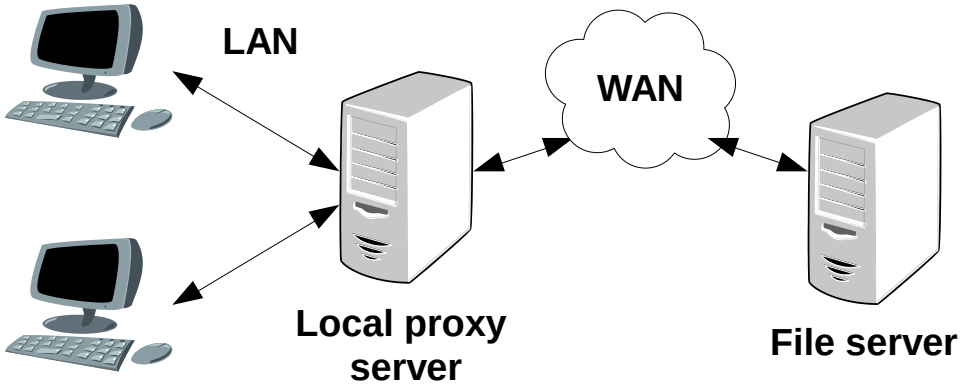

Fig. 13 Local proxy caching

5 Future work

In our future work, we will aim at development of caching algorithm for mobile clients. Caching algorithms are usually based on statistical information made from client's requests from the past. The best caching algorithms seem to be LRU or LFU. A problem while using these two algorithms is that a new file incoming to the cache has no statistical info. The old files in the cache are more protected from throwing out from the cache than the new files.

In this case, we will adopt storing statistical information on a server side. The server will store hits for reading and writing for each file. This information will be provided as metadata when the file is requested by a client. Based on this information we will develop a mathematical model. This model will calculate statistical information for each new file and we will decide whether to store the new file in the cache or not. The new files will have the same chance to be stored in the cache as the cached files.

Another problem while using the cache is data consistency. Consistent state means that the files in the cache and on the server have the same timestamp. On the server side we use logical clock per each file. On the client side we will adopt algorithms which will periodically obtain new versions of cached files from the server. If there is a new version of the file then this will be downloaded and stored in the cache. For reducing network communication, we will develop mathematical model which will separate the cached files into groups based on number of writing hits from the server. For each group of files, we will use different period time for obtaining new file versions. This approach supposes that we will obtain files which are frequently updated on the server side more often.

We will also adopt so called off-line operation. Mobile devices have no guarantee of having permanent network access. For this case, we will provide the user the cached files. The user can work with cached files without knowing that the files are stored on her/his device. When the client is connected again to the server, new versions of the cached files will be uploaded to the server and new versions of the files will be downloaded from the server to the cache. If there are any conflicts between files, the user must solve them manually.

In the mobile communication, we can use several technologies for data transfer. There exist slow technologies (download speed: GPRS at the maximum 80 kbit/s; EDGE at maximum 236.8 kbit/s) and also fast technologies (HSDPA at maximum 21.1 Mbit/s; Wi-Fi at maximum 54Mbit/s). Slow technologies are usually available everywhere. Fast technologies are usually available in the cities. In our future work, we will attempt to make algorithms which will choose the fastest

technology at a given moment. This algorithm will also periodically check, whether there is any faster technology. Based on another mathematical model, we will check for better technology more often, if the current communication technology is slow.

For evaluation of our ideas, we will use KIV-DFS. KIV-DFS is a distributed file system which is being developed at the Department of Computer Science and Engineering (Katedra Informatiky a Výpočetní techniky), University of West Bohemia. This distributed file system considers usage of mobile devices, and adopts algorithms for accessing data from these devices.

Bibliography

[1] Andrew S. Tanenbaum and Maarten Van Steen, *Distributed Systems : Principles and Paradigms*. Upper Saddle River: Prentice Hall, 2002.

[2] (2002) Transparency in Distributed Systems. [Online]. http://crystal.uta.edu/~kumar/cse6306/papers/mantena.pdf

[3] Philippas Tsigas, "AFS Report," Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, Lecture 2010.

[4] John H. Howard, "An Overview of the Andrew File System," in *Proceedengs of the USENIX Winter Technical Conference*, Dallas TX, 1988.

[5] Luboš Matějka, "Distribuované souborové systémy (Distributed File Systems)," in *Počítačové architektury & diagnostika : Pracovní seminář pro studenty doktorského studia : Lázně Sedmihorky*, 2005, pp. 125-128.

[6] T.Y.C. Woo and S.S Lam, "Authentication for distributed systems," in *Computer , vol. 25, no. 1*, 1992, p. 39.

[7] Rainer Többicke, "Distributed File Systems: Focus on Andrew File System/Distributed File Service (AFS/DFS)," in *Mass Storage Systems, 1994. 'Towards Distributed Storage and Data Management Systems.' First International Symposium. Proceedings., Thirteenth IEEE Symposium on*, Annecy, France, 1994, pp. 23-26.

[8] Jan Steuer. (2008) RPC, NFS, Automount. [Online]. http://www.fi.muni.cz/~kas/p090/referaty/2008-jaro/st/nfs.html

[9] B. Liskov, R. Gruber, P. Johnson, and L. Shira, "A replicated Unix file system," in *Management of Replicated Data, 1990. Proceedings., Workshop on the*, Houston, 1990, p. 11.

[10] M. Tim Jones. (2008, Jun) Anatomy of Linux journaling file systems. [Online]. http://www.ibm.com/developerworks/library/l-journaling-filesystems/index.html

[11] Holger Brueckner, Benny Chuang, Tiemo Kieft, and Steven McCoy. (2004, Sep) Gentoo Linux OpenAFS Guide. [Online]. http://61.153.44.88/gentoo/resources/document-listing/openafs.html

[12] Mahadev Satyanarayanan, "Scalable, secure, and highly available distributed file access," in *Computer*, 1990, pp. 9 - 18, 20-21.

[13] M. van Steen and G. Pierre, "Replication for Performance: Case Studies," in *Lecture Notes in Computer Science, Volume 5959.*, 2010, pp. 73-89.

[14] Microsoft. (2011, Sep) DFSR Overview. [Online]. http://msdn.microsoft.com/en-us/library/windows/desktop/bb540025(v=vs.85).aspx

[15] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," , Incline Village, NV, 2010, pp. 1-10.

[16] S. Abdalla, I. Ahmad, Ewe Hong Tat, Gim Aik Teh, and Yong Lee Kee, "Towards Achieving a Highly Available Distributed File System," in *Advanced Communication Technology, The 9th International Conference on*, Gangwon-Do, 2007, pp. 2056-2060.

[17] Xin Sun, Jun Zheng, Qiongxin Liu, and Yushu Liu, "Dynamic Data Replication Based on Access Cost in Distributed Systems," in *Computer Sciences and Convergence Information Technology, 2009. ICCIT '09. Fourth International Conference on*, Seoul, 2009, pp. 829-834.

[18] Xiaodong Li and Chang Liu, "Towards a Reliable and Efficient Distributed Storage System," in *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on*, 2005, pp. 301c - 301c.

[19] D. Peric, T. Bocek, F.V. Hecht, D. Hausheer, and B. Stiller, "The Design and Evaluation of a Distributed Reliable File System," in *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, Higashi Hiroshima, 2009, pp. 348-353.

[20] (2008) CloudStore. [Online]. http://kosmosfs.sourceforge.net/features.html

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google file system," in *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003, pp. 29-43.

[22] (2010) Introduction to Gluster Versions 3.0.x. [Online]. http://download.gluster.com/pub/gluster/documentation/Introduction_to_Gluster.pdf

[23] Jun Lu, Bin Du, Yi Zhu, and DaiWei Li, "MADFS: The Mobile Agent-Based Distributed Network File System," in *Intelligent Systems, 2009. GCIS '09. WRI Global Congress on*, Xiamen, 2009, pp. 68-74.

[24] Lihua Yu, Gang Chen, Wei Wang, and Jinxiang Dong, "MSFSS: A Storage System for Mass Small Files," in *Computer Supported Cooperative Work in Design, 2007. CSCWD 2007. 11th International Conference on*, Melbourne, Australia, 2007, pp. 1087-1092.

[25] Lei Wang and Chen Yang, "TLDFS: A Distributed File System based on the Layered Structure," in *NPC '07 Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, Dalian, China, 2007, pp. 727-732.

[26] Ananth Devulapalli and Pete Wyckoff, "File Creation Strategies in a Distributed Metadata File System," in *2007 IEEE International Parallel and Distributed Processing Symposium, 2007*, Long Beach, CA, USA, 2007, p. 105.

[27] P. Carns et al., "Small-file access in parallel file systems," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, Rome, Italy, 2009, pp. 1-11.

[28] Bin Cai, Changsheng Xie, and Guangxi Zhu, "EDRFS: An Effective Distributed Replication File System for Small-File and Data-Intensive Application," in *Communication Systems Software and*

Middleware, 2007. COMSWARE 2007. 2nd International Conference on, Bangalore, 2007, pp. 1-7.

[29] J. Stender, B. Kolbeck, M. Hogqvist, and F. Hupfeld, "BabuDB: Fast and Efficient File System Metadata Storage," in *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*, Incline Village, NV, 2010, pp. 51-58.

[30] Benjamin Reed and Darrell D. E. Long, "Analysis of caching algorithms for distributed file systems," in *ACM SIGOPS Operating Systems Review, Volume 30 Issue 3*, New York, NY, USA, 1996, pp. 12-17.

[31] B. Whitehead, Chung-Horng Lung, A. Tapela, and G. Sivarajah, "Experiments of Large File Caching and Comparisons of Caching Algorithms," in *Network Computing and Applications, 2008. NCA '08. Seventh IEEE International Symposium on*, Cambridge, MA, 2008, pp. 244-248.

[32] K.W. Froese and R.B. Bunt, "The effect of client caching on file server workloads," in *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on*, Wailea, HI , USA, 1996, pp. 150-159.

[33] A. Ermolinskiy and R. Tewari, "C2Cfs: A Collective Caching Architecture for Distributed File Access," in *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, Seoul, 2009, pp. 642-647.

[34] Lamprini Konsta and Stergios V. Anastasiadis, "Hades: Locality-aware Proxy Caching for Distributed File Systems," 2009.