ALGORITHMS FOR INCREASING PERFORMANCE IN DISTRIBUTED FILE SYSTEMS

Pavel BŽOCH, Jiří ŠAFAŘÍK
Department of Computer Science and Engineering, Faculty of Applied Sciences,
University of West Bohemia, Univerzitní 22, 306 14 Pilsen, Czech Republic,
tel.: +420 377 632 414, 632 439, e-mail: pbzoch@kiv.zcu.cz; safarikj@kiv.zcu.cz

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 systems (DFS) provide many advantages such as reliability, scalability, security, capacity, etc. In the paper, 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 increasing performance will be discussed. These include the organization of data and metadata storage, usage of caching, and design of replication algorithms. This paper provides overview of existing algorithms which are used in DFS. Described algorithms can be used as a basis for any future work.

Keywords: *distributed file systems, file replication, caching, performance*

1. INTRODUCTION

Distributed systems (DS). Modern scientific computations require powerful hardware. One way of getting results in scientific work faster is purchasing new hardware over and over again. Buying powerful hardware is, however, not a cheap solution. Another way is using distributed systems, where the need for performance is spread over several computers.

In distributed systems, several computers are connected together usually by LAN. In the client's view, all these computers act together as one computer. This concept brings many advantages. Better performance can be achieved by adding new computers to the existing system. If any of the computers crash, the system is still available.

Using DS brings several problems too. In the distributed systems, we have to solve synchronization among computers, data consistency, fault tolerance, etc. There are many algorithms which solve these problems. Some of them are described in [1].

Distributed file systems (DFS). Distributed file systems 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*.

The data in a DFS are then 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*. The data replication also increases system performance – a client can download a file from the node which is the most available one at a given moment.

Files can also be transparently moved among nodes. This is typically invoked by an administrator and it is done for improving 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 also 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*.

2. SUMMARY OF TRADITIONAL DFS

This section will describe traditional DFS. There are many DFS, some of which are commercial (like AFS), and others free (like OpenAFS, NFS, Coda and Samba). This section will describe traditional DFS like NFS4, OpenAFS, Coda and SMB. Many of the new DFS extend or are based on these traditional DFS.

2.1. OpenAFS

AFS (Andrew File System) was originally created at Carnegie Mellon University; later it was a commercial product supported by IBM. Now it is being developed under a public license.

AFS has 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-tree, 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 tree. Information about the whole system is stored in a special database server [3].

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. For achieving better performance of read-only files, *replica servers* can be used. When the other servers are overloaded, the replica server provides files to the clients instead of these servers.

AFS is a very stable and robust system and it is often used at universities. AFS uses Kerberos [4] as an authentication and authorization mechanism. More information about AFS can be found in [5].

2.2. NFS4

NFS (Network File System) is an internet protocol which was originally created by Sun Microsystems in 1985, and was made 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 a part of the Linux kernel. The Kerberos system is used for user identification. System performance is increased by using a local client cache. NFS has two main elements: a client and a data server.

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 [3].

In NFS, there usually exists an *automounter* on client side [6]. 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.

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. Coda supports off-line working, which means that cached files are available even after disconnection from the server. 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 uses read-only replicas which 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

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 and uses the operating system's file access rights.

3. DATA AND METADATA STORAGE ORGANIZATION

This section will describe modern trends in DFS with a focus on data storage and metadata storage. Data storage is used for storing file content. Metadata storage usually stores file attributes and links to the file content in the data storage.

3.1. 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. The whole uploading process is depicted in Figure 1.

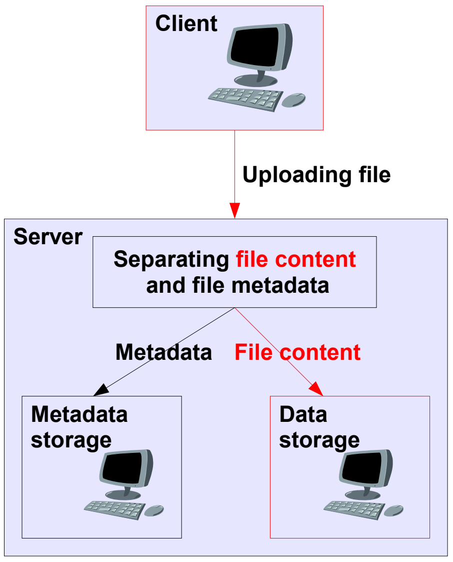

Fig. 1 File uploading process (file content)

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 [7], the RaiserFS 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 [7]. 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 the existing data organization (file system) of a hard disk is not suitable for files which will be stored there.

While using method from [8], 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 servers for recording weather a block is used or not [8]. The block number is used to number the inode, so every inode use the block exclusively. Every single inode can be locked without increasing the total amount of locks. Distribution of the inodes also makes the system more expandable because the distribution makes it possible for the number of inodes to increase or decrease dynamically [8].

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 have 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 → connecting metadata with the file handle. Both, creating the metadata record and storing content, are slow operations. According to [9] and [10], these slow operations can be accelerated.

Paper [9] presents increasing performance by making changes in an upload protocol. These changes can be made in different ways:

Compound operations. In compound operations, we suppose that the steps in upload protocol are independent from the others. Thus we can do these steps 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 a case of using leased handles, 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 of a metadata record and a +file uploading can be done parallel.

Paper [10] presents increasing performance by using another methods. These methods presume uploading huge amount of small files. Method pre-creating file object is similar to [9]. 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. The client application can create a metadata record 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 approach 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 approach saves one message.

Both [9] and [10] demand cooperation between the file storage nodes and the metadata storage nodes.

Papers [7], [8], [9] and [10] 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 approach 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. This system also provides file replication. The entire system is described in [11]. The P2P system architecture is depicted in Figure 8.

3.2. Metadata storage

Metadata are a specific type of data which give us information about a certain item's content. In DFS, metadata are 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 2). 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 usually are two types of locks: a shared lock for file reading and an exclusive lock for writing or updating file content. Metadata are usually stored in a database or in tree.

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 in [12] and [13]. 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. Node corresponding to the file must also have a link to the file content in the data storage. When the client requests a file, the algorithm described in [12] returns links to all data storages where file replicas are stored. The algorithm described in [13] returns a link to the data storage which is the closest to the client. The metadata node keeps a list of available data nodes by receiving heartbeat messages from these nodes.

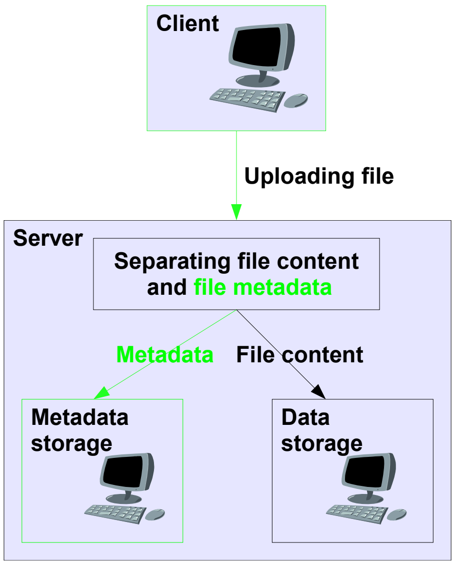

Fig. 2 File uploading process (metadata)

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 [14]. Paper [14] 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 [14]. 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 wasn't 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 Figure 3.

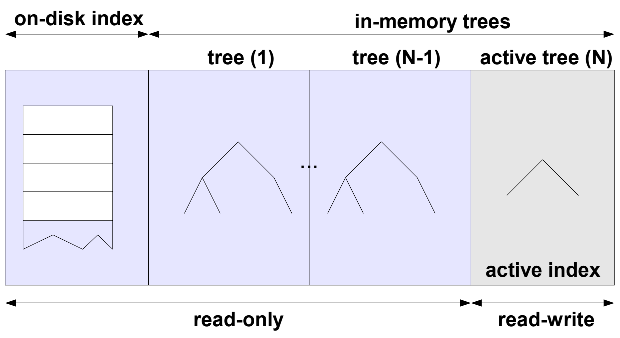

Fig. 3 Database with N in memory trees and on-disk index [14]

4. CACHING AND REPLICATION ALGORITHMS

The previous section describes how the choice of data and metadata storage can influence DFS performance. This section will describe caching and replication algorithms which can also increase DFS performance.

4.1. Caching

A cache in the computer system is a component which stores data that 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. There are several caching policies which try to predict future requests.

Caching policies are used 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. Most of caching policies are described in [15]. 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 [15], the most effective replacement policy is LRU.

The *LRU* (Last 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 again in the future.

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. Paper [16] extends existing LRU and LFU policies with a Size and Threshold policy. An 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 [16] is, again, LRU with no extension.

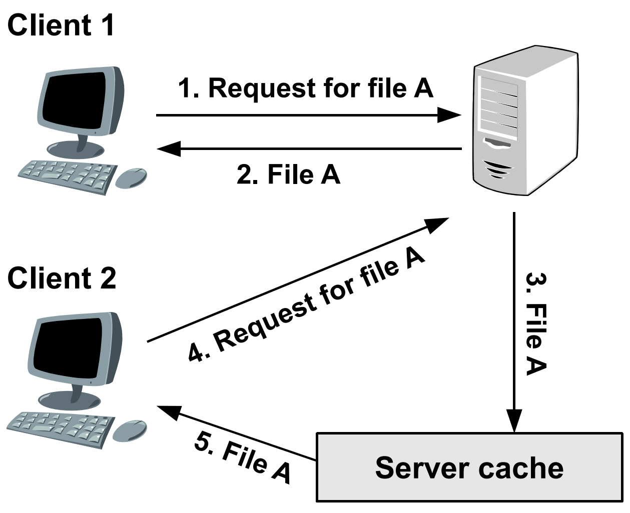

Fig. 4 Server-side caching

A cache can used be 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 4). At a server side, cache can be stored in-memory or on separate machine. The client stores in the cache the data which may be requested again in the future (see Figure 5). Client cache is usually stored in-memory. 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 [17].

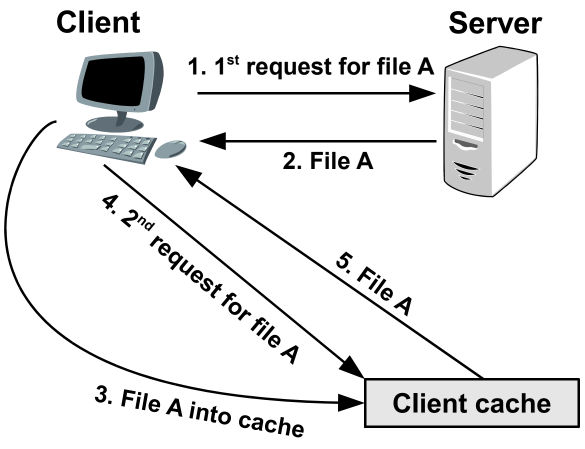

Fig. 5 Client-side caching

Paper [18] presents a decentralized collective caching architecture. In this paper, 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 to 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 algorithm 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 [19]. A proxy cache stores files which are requested by clients. 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. 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. Proxy caching is depicted in Figure 6.

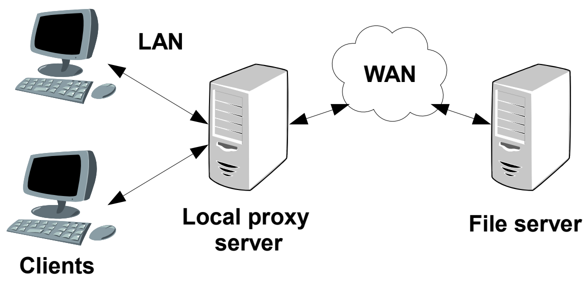

Fig. 6 Local proxy caching

4.2. Replication

Replication in a DFS is a process whereby the original file is copied to other servers in the DFS. The original file is usually called *the primary replica* or *master replica;* other copies are called *replicas*. A replication algorithm (or strategy) describes which data will be replicated, as well as how, when, and where the replicated file should take place [20].

Replication can be used for achieving better performance, availability or fault tolerance. All these three requirements use slightly different algorithms for choosing the file and the place for the replication.

We will focus on replication for achieving better performance. In this 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.

Replication can be done statically (administratively) or dynamically. In a static replication system, the administrator marks storage where the primary replica is placed. The administrator then defines the number of replicas and the replicas' placement. In this case, the administrator predicts which files will probably be the most used in the future.

Another method of file replication is dynamic replication (see Figure 7). Dynamic file replication is described in [21] and [22]. Both of these papers represent dynamic replication based on statistical information.

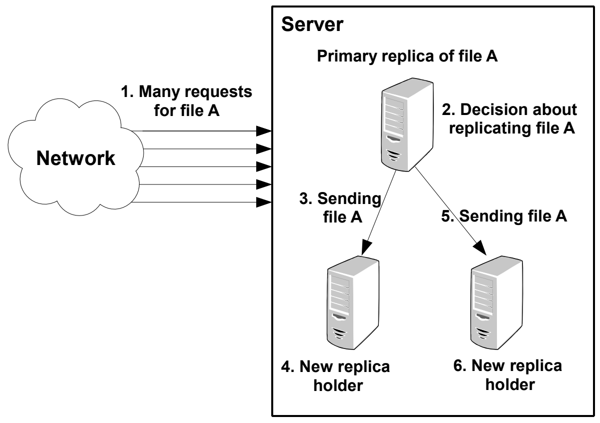

Fig. 7 Dynamic file replication

Paper [21] presents the Criteria Based Replication (CBR). This model uses two main algorithms for achieving high availability and reliability. The first algorithm is the Replica Placement algorithm. This algorithm collects information about physical location of the server, load of the server, etc. The second algorithm is the Primary-copy Assignment. 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 (CBRP) algorithm monitors each criterion individually [21] .Then, it periodically calculates a result for each criterion. If this result exceeds a threshold value, the file is replicated.

In the Criteria Based Primary-copy Assignment (CBPA) 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 [21]. 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 [22] 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 [22].

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 [22]. 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.

Another dynamic replication is described in [11]. This paper uses P2P architecture where fragments of files are stored on several peers. The system architecture is depicted in Figure 8. The peer closest to the file fragment ID is responsible for that fragment and has to check periodically if enough replicas are available [11]. If there are not enough replicas in the system, the peer can replicate fragments.

File replication for increasing performance is also used in other DFS. CloudStore [23] typically uses 3-way file replication (files are typically replicated to three nodes). 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 is derived from the Google File System [24].

GlusterFS [25] uses three file replication options. The first option is a file distribution over mirrors. This means that each storage server is replicated to another storage server. Other ways of storing files are file distribution to one node or file stripping over nodes. These two solutions are less reliable.

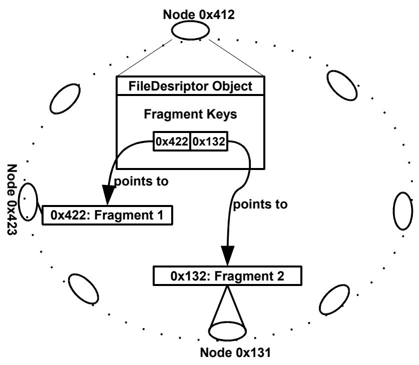

Fig. 8 P2P system architecture in [11]

5. CONCLUSION

This paper provides state of the art in DFS in increasing performance in DFS. All algorithms described in sections 3 and 4 were developed in recent times. This paper summarizes these algorithms and can be used as a basis for future research. System performance can be increased by choosing a suitable file system. If there is no suitable file system, a new one can be developed. Another way of increasing performance is by accelerating I/O operations, which are the bottleneck of system performance. To achieve better performance a metadata storage scheme is also important. Metadata operations make up over half of the workload of the DFS. Other two methods, caching and file replication, can be added into the system later. Caching methods can increase system performance by predicting future requests. File replication can increase system performance by spreading the system work-load to more servers. Caching and replicating algorithms can greatly increase system performance, but both of these algorithms do not always work reliably. On the one hand, these algorithms may increase system performance; on the other hand, if they are incorrectly set, system performance can be decreased.

ACKNOWLEDGMENTS

This work is supported by the Ministry of Education, Youth, and Sport of the Czech Republic – University spec. research – 1311. We thank Ladislav Pesička and Luboš Matějka, Doctoral students, Department of Computer Science and Engineering, University of West Bohemia, for their support and ideas.

REFERENCES

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

[2] Transparency in Distributed Systems, 2002. http://crystal.uta.edu/~kumar/cse6306/papers/manten a.pdf

[3] MATĚJKA, L.: Distributed File Systems, in *Computer Architecture and Diagnostic: Workshop for Doctoral Students: Lázně Sedmihorky*, 2005, pp. 125–128.

[4] WOO, T.Y.C. – LAM, S.S.: Authentication for Distributed Systems, *Computer*, Vol. 25, No. 1, pp. 39–52, January 1992.

[5] TÖBBICKE, R.: 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.

[6] LISKOV, B. – GRUBER, R. – JOHNSON, P. - SHIRA, L.: A replicated Unix file system, in *Management of Replicated Data, 1990, Proceedings, Workshop on the*, Houston, 1990, pp. 11–14.

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

[8] Lei Wang – 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.

[9] WYCKOFF, P. – DEVULAPALLI, A.: File Creation Strategies in a Distributed Metadata File System, in *2007 IEEE International Parallel and Distributed Processing Symposium, 2007*, Long Beach, CA, USA, 2007, pp. 105.

[10] CARNS, P. et al.: Small-file Access in Parallel File Systems," in *Parallel & Distributed Processing, IPDPS 2009. IEEE International Symposium on*, Rome, Italy, 2009, pp. 1–11.

[11] PERIC, D. – BOCEK, T. – HECHT, F. V. – HAUSHEER, D. – STILLER, B.: 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.

[12] Bin Cai – Changsheng Xie – Guangxi Zhu: EDRFS: An Effective Distributed Replication File System for Small-File and Data-Intensive Application, in *Communication Systems Software and Middleware, COMSWARE 2007, 2nd International Conference on*, Bangalore, 2007, pp. 1–7.

[13] SHVACHKO, K. – HAIRONG KUANG, S. – RADIA, S. – CHANSLER, R.: The Hadoop Distributed File System, Incline Village, NV, 2010, pp. 1–10.

[14] STENDER, J. – KOLBECK, B. – HƠGQVIST, M. – HUPFELD, F.: 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.

[15] REED, B. – LONG, D.D.E.: Analysis of Caching Algorithms for Distributed File Systems, in *ACM SIGOPS Operating Systems Review, Vol. 30, Issue 3*, New York, NY, USA, 1996, pp. 12–17.

[16] WHITEHEAD, B. – CHUNG-HORNG LUNG – TAPELA, A. – SIVARAJAH, G.: Experiments of Large File Caching and Comparisons of Caching Algorithms, in *Network Computing and Applications, NCA '08, Seventh IEEE International Symposium on*, Cambridge, MA, 2008, pp. 244–248.

[17] FROESE, K. W. – BUNT, R. B.: The Effect of Client Caching on File Server Workloads," in *System Sciences, Proceedings of the Twenty-Ninth Hawaii International Conference on*, Wailea, HI, USA, 1996, pp. 150–159.

[18] ERMOLINSKIY, A. – TEWARI, R.: C2Cfs: A Collective Caching Architecture for Distributed File Access, in *High Performance Computing and Communications, HPCC '09, 11th IEEE International Conference on*, Seoul, pp. 642–647.

[19] KONSTA, L. – ANASTASIADIS, S. V.: Hades: Locality-aware Proxy Caching for Distributed File Systems, 2009.

[20] van STEEN, M. – PIERRE, G.: Replication for Performance: Case Studies, in *Lecture Notes in Computer Science,* Vol. 5959, 2010, pp. 73–89.

[21] ABDALLA, S. – AHMAD, I. – Ewe Hong Tat – Gim Aik The – 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.

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

[23] CloudStore. http://kosmosfs.sourceforge.net/features.html

[24] Sanjay Ghemawat Howard Gobioff – 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.

[25] Introduction to Gluster Versions 3.0.x. http://download.gluster.com/pub/gluster/documentation/IntroductiontoGluster.pdf

Received January 18, 2012, accepted May 24, 2012

BIOGRAPHIES

Pavel Bžoch was born on 4.11.1985 in Prachatice, Czech Republic. In 2010, he graduated with distinction at the Department of Computer Science and Engineering at University of West Bohemia. Nowadays, he is a PhD. student at the Department of Computer Science and Engineering. His research covers distributed computing and distributed file systems. His supervisor is prof. Šafařík.

Jiří Šafařík was born in Kromeriz, Czech Republic. He received his Ph.D. degree from Slovak University of Technology in 1984. Currently, he is professor in Department of Computer Science and Engineering at Faculty of Applied Sciences of University of West Bohemia. His research covers distributed systems, distributed and parallel simulation.