Politechnika Wrocławska

Wydział Informatyki i Zarządzania
kierunek studiów: Informatyka
specjalność: Inżynieria oprogramowania

Praca dyplomowa - magisterska

SAMOORGANIZUJĄCY SIĘ ROZPROSZONY SYSTEM PLIKÓW

SELF-ORGANIZING DISTRIBUTED FILE SYSTEM

Roman Kierzkowski

słowa kluczowe:
rozproszony system plików, samoorganizacja, replikacja, algorytm rozmieszczania replik, modele spójności, komunikacja grupowa, wirtualna synchroniczność, algorytmy mrówkowe, plotkowanie

krótkie streszczenie:
Praca bada możliwość wykorzystania zjawiska samoorganizacji do stworzenia algorytmu rozmieszczania replik dla potrzeb rozproszonego systemu plików. Studium na temat istniejących systemów oraz algorytmów rozmieszania replik zawarte w opracowaniu posłużyło zaprojektowaniu i implementacji Samoorganizującego się Rozproszonego Systemu Plików (SoDFS). System implementuje algorytm zaproponowany przez autora. W pracy znalazło się również omówienie zagadnień teoretycznych związanych z projektowaniem rozproszonego systemu plików. Opracowanie podsumowuje badanie stworzonego systemu w środowisku emulującym działanie sieci rozległej.

Promotor:	dr inż. Dariusz Konieczny	.....	.....
	imię i nazwisko	*ocena*	*podpis*

Wrocław 2009

Kochanym Rodzicom

To My Dear and Loving Parents

The Acknowledgements

I would like to express my gratitude to all the people, who helped me to write this thesis. Especially, I would like to thank Dirk Nowotka, Ph.D., who was my advisor during the time I spent at the Stuttgart University, for his continuous support, commitment, time, patience and accurate advice. I also would like to thank professor Kurt Rothermel from Stuttgart University for making the NET emulation testbed accessible to me. A special thanks goes to Andreas Grau, M.S. from Stuttgart University for adapting the testbed for my needs and for his patient help during the configuration process. Last, but not least, I thank my friend Kristyn Orgill, who greatly contributed to the improvement of language aspects of my work.

Contents

Abstract

This work investigates the prospect of utilizing the phenomenon of self-organization to create a replica placement algorithm for the needs of a distributed file system. A system exhibits property of self-organization if it reaches the internal arrangement as a result of autonomous decisions of its parts. The study over the existing distributed file systems and the replica placement algorithms extracted the desired properties, which are absent in the accessible solutions. The gathered knowledge helped create a Self-organizing Distributed File System (SoDFS). The system implements a Self-organizing Replica Placement Algorithm (SoRPA) proposed by the author. The algorithm obtains the desired replica arrangement as an outcome of a replication request and orders interchanged between the system nodes. The thesis also contains the discussion of theoretical aspects related to designing of a distributed file system, such as consistency models, the reliable multicast protocols and the virtual synchrony model. The elaboration is summarized with the evaluation of the created system in the environment emulating the wide area network.

Streszczenie (Abstract in polish)

Praca bada możliwość wykorzystania zjawiska samoorganizacji do stworzenia algorytmu rozmieszczania replik dla potrzeb rozproszonego systemu plików. System posiada właściwość samoorganizacji, jeśli uzyskuje wewnętrzne uporządkowanie w efekcie autonomicznych decyzji swoich części. Studium na temat istniejących rozproszonych systemów plików oraz algorytmów rozmieszania replik, zawarte w opracowaniu, pozwoliło wyłonić pożądane właściwości, których nie mają dostępne rozwiązania. Wiedza ta posłużyła w stworzeniu Samoorganizującego się Rozproszonego Systemu Plików (SoDFS). System implementuje Samoorganizujący Algorytm Rozmieszczania Replik (SoRPA) zaproponowany przez autora. Algorytm uzyskuje pożądane uporządkowanie replik na skutek wymian próśb i rozkazów replikacji wysyłanych wzajemnie między węzłami systemu. W pracy znalazło się również omówienie zagadnień teoretycznych związanych z projektowaniem rozproszonego systemu plików, takich jak modele spójności, protokoły niezawodnej komunikacji grupowej oraz model synchroniczności wirtualnej. Opracowanie podsumowuje badanie stworzonego systemu w środowisku emulującym działanie sieci rozległej.

1. Introduction

The numerous *distributed file systems* (*DFS*) were developed over the years. The initial research was focused on the semantic of a remote file access. The noticeable work in that field was done over *NFS* and *CIFS*. The next step of development introduced the replication to the systems. The files (*Coda*) or their parts (*GFS*, *Wuala*) are present in many copies on the distributed servers to improve the efficiency of the solution or make it fault tolerant. The existing systems, reviewed in the work, present different approaches to the replication. In a system like *MS DFS* or *Coda* the replicas are created by the system administrator. He decides, where copies of a directory or a volume should be placed. In *Wuala* the file chunks are located at random peers. The *GFARM*, instead of moving files between nodes, migrates the computations closer to the data. The *GFS* has a central algorithm, which autonomously decides where the chunk replicas should be placed. The algorithm optimizes disk space utilization. Since the *replica placement problem* was proved to be NP-complete, the numerous *replica placement algorithms* (*RPA*) were proposed to approximate optimal solution. Some of them found their application in the *Content Delivery Networks* (*CDN*).

A review of the existing replica placement algorithms showed that they are burden with weaknesses. The heuristics like *Greedy-Global* require a global knowledge about the system – the structure of a network connecting the peers, the file usage statistics from every node etc. This kind of information is not always accessible or gathering it might be expensive. Some of the simple heuristics, like *Random*, omit information about the demand for certain files in a distinct part of the network. As a consequence they also improve the performance in the parts that do not require it. Others, like *Popularity* heuristic, do not take into account the difference between the reads and the writes. The cost of a write operation is larger than a cost of read operation, because it involves update propagation to all replicas. The more complex algorithms (*ADR*), which provide the read/write optimization and utilize only the local knowledge, work over some specific overlay structures, which entail an additional cost for building them.

The goal of the work was to create a distributed file system and check the prospect of utilizing a self-organization phenomenon in its replica placement algorithm. The system exhibits a self-organization property if it reaches the desired global arrangement as a result of autonomous decision of its parts. The phenomenon was observed in nature and became the inspiration for the numerous algorithms, like the ant colony optimization algorithm or update dissemination protocol based on gossiping. The desired RPA should overcome the weaknesses of the existing solutions. It should utilize the knowledge about the local demands to reduce the client perceived latency in the given part of the system and should simultaneously achieve the tradeoff between improving the reads and writes.

As a result of the work, the *Self-organizing Distributed Files System* (*SoDFS*) was created. It implements the *Self-organizing Replica Placement Algorithm* (*SoRPA*) proposed in the thesis. The algorithm establishes the replica placement as an outcome of replication request and orders interchanged between the nodes – no coordinator is present or being elected in the process. Although the work is focused on the replication, it also presents the numerous problems related with the implementation of a DFS. The existence of multiple users modifying the same file causes the consistency problems, therefore the work describes the consistency models and communication primitives allowing their realization. The presence of failures necessitates the implementation of the virtual synchrony model. The implementation of movable replicas is a challenge, hence the replica state model is proposed. Thanks to used tools (*JGroups*, *JLAN*) and modular architecture, the SoDFS is a flexible platform for testing RPA, which can be easily extended to be fault tolerant and secure.

The SoDFS evaluation was performed in the emulation testbed provided by the Universität Stuttgart, which made possible to test the solution in the wide area network.

2. Existing systems review

2.1. Sun NFS

The *Network File System* (*NFS*) [30] is a set of protocols that provide clients with a model of a remote file system. NFS is a product of Sun Microsystems. The second version of NFS was the first version publicly available as a part of SunOS 2.0. The current fourth version is a deep revision of the third version including changes in security and an adjustment to the requirements of the Internet.

NFS implements a *remote access model*. It means that clients are operating on remote files by calling operations from the file system interface, which are then executed on files by the server. NFS clients use system calls to execute operations on the *Virtual File System* (*VFS*), which is a standard interface for many (distributed) file systems.

The server *exports* a folder that the client can mount in any place of his own files system tree. The client can also mount the subfolders of the exported folder. Therefore the client perceives files as his own local files. The operations executed on the VFS are then transferred to the server by the (Open Network Computing) RPC executions. This procedure stays transparent for the client.

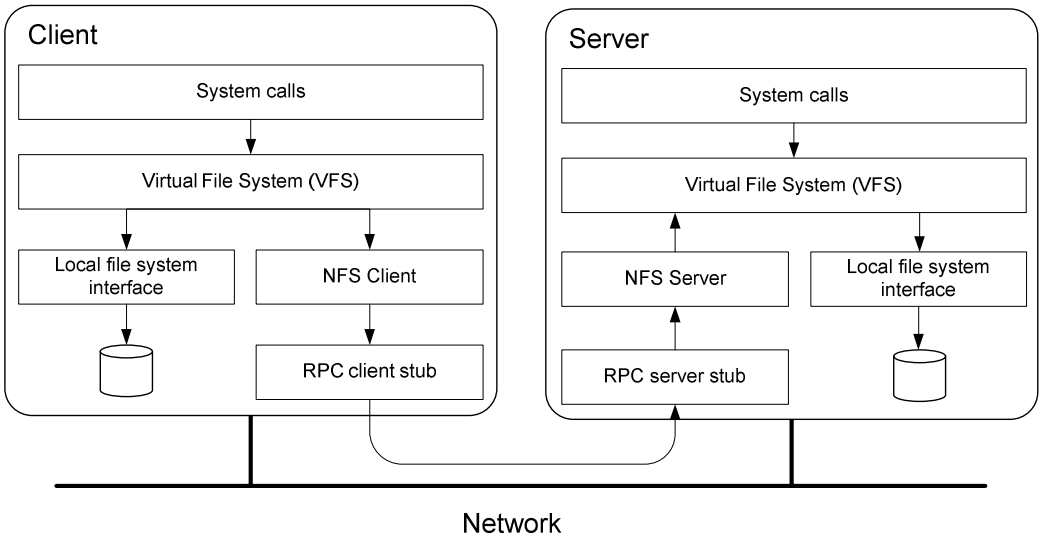


Fig 2.1. NFS architecture

NFS implementation may assure *UNIX semantic* of concurrent access to the file. It means that the *read* operation after a successful *write* operation returns the value that has been written. Although, many implementations of the NFS version 3 caches local copies of the remote file by the client's side, without assuring any kind of consistency.

NFS version 4 provides two mechanisms of file caching. The first kind assures *session semantic*. It means that *open* operation causes file caching (or cache validation if the file was previously cached) on the client's side. Then the client can read and write the file locally but changes are transferred to the server's copy when the client is closing the file.

The second mechanism is called *open delegation.* The server can delegate to the specific client machine rights for opening and closing a file. All clients that are on that machine are served locally if they want to operate on this file. If the file was delegated with rights to write, other requests for write which come from other machines will be rejected. If the file was delegated with rights to read all write requests are held by the server. The server can withdraw delegation. To make it possible it must hold information about the client where rights were

delegated. Because the NFS server in version four is not stateless anymore as it has been in previous versions.

NFS version 4 provides small support for replication. Each file should have FS_LOCATION an attribute that contains possible localization of file systems that contains that file.

2.2. CIFS and SMB

CIFS (Common Internet File System) is an application layer protocol based on SMB (Server Message Block) protocol created by IBM and considerably modified by Microsoft. It has similar functionality to NFS. It is used to share files, printers and serial ports.

2.3. Microsoft Distributed File System

The *Distributed File System* [40] is a solution contained in the Microsoft Windows Server 2003 R2 (enable also in Windows NT 4.0 and Windows 2000). Because of its confusing name in this paper we will use the name *Microsoft Distributed File System* (*MS DFS*) instead. It is based on two technologies *DFS Namespace* and *DFS Replication*:

- *DFS Namespaces* allows the grouping of shared folders (shared through SMB/CIFS protocol) located on different machines into one namespace which is presented to the user as a virtual tree of folders.
- *DFS Replication* is a state-based and multimaster replication engine. It supports replication scheduling and bandwidth throttling. The *Remote Differential Compression* (*RDC*) protocol enables DFS Replication to replicate only the changes when files are synchronized. One of its functions called *Cross-File RDC* reduces use of bandwidth during the replication of new files. It uses heuristic on the target server to identify files similar to the replicated file. Only the parts of the original file that are not contained in those files must be sent during the replication over WAN.

Although both technologies may be used separately, when combined they provide solutions for the medium and large enterprise systems.

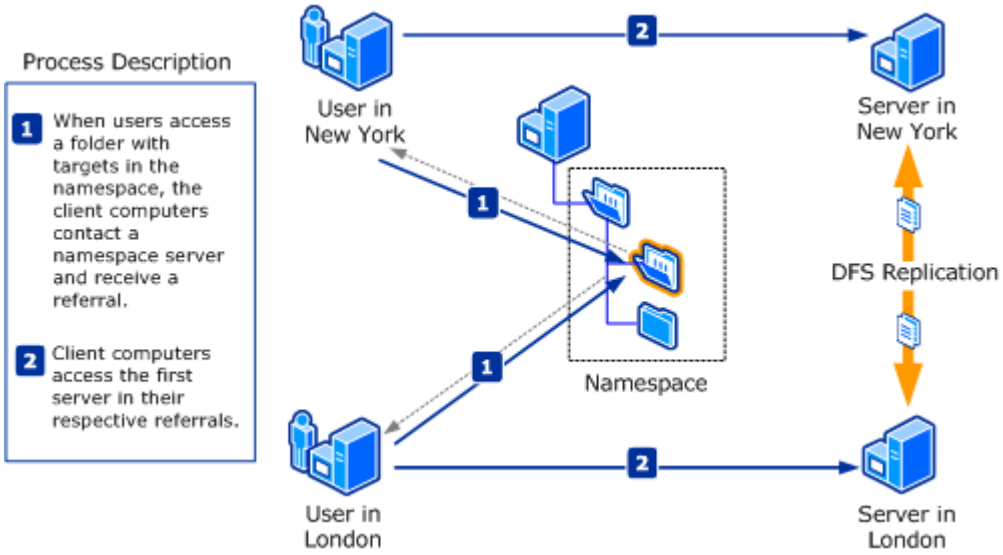


Fig 2.2. DFS Namespace and DFS Replication [figure source: Microsoft Tech]

The Fig 2.2. presents how DFS Namespace and DFS Replication works together:

1. When a user attempts to open a folder in the namespace, his computer contacts with the namespace server and receives a referral that contains a list of servers that host shared the folder.
2. The client computer contacts with one of the servers in the referral, located in the client's own site or chosen according to the administrator defined target priority.

The replication is configured manually by an administrator. The synchronization of distributed replicas is performed periodically. MS DFS requires the file to be closed during the synchronization process.

2.4. Coda

Coda [30] The file system has been developed at Carnegie Mellon University (CMU) since 1987. The goal of the system was to ensure great availability. To achieve it Coda uses advanced methods of caching and server replication that allows disconnected operation and continued operation during partial network failures.

Coda inherits its architecture form *AFS* (*Andrew File System*) which was also developed at CMU. The system was developed to be available for the whole university network – approximately 10000 workstations. Machines are divided into two groups:

- *Vice* – centrally administrated file servers,
- *Virtue* – work stations that allow users access to the file system.

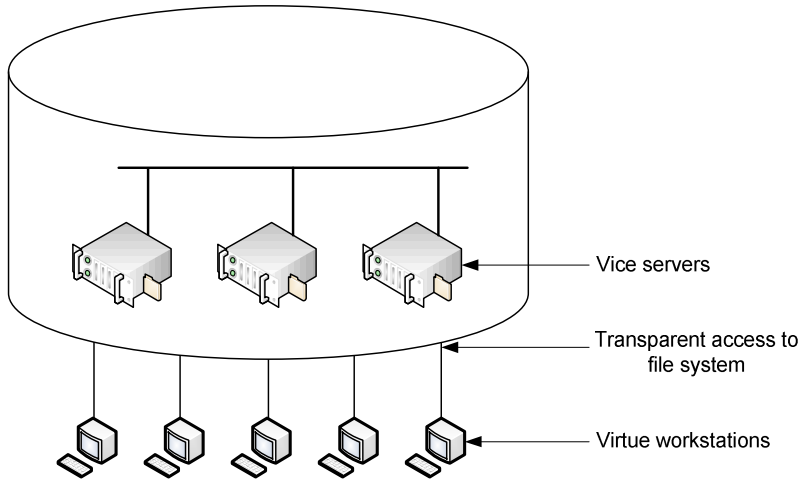


Fig 2.3. Organization of Coda

Each Virtue workstation runs the *Venus* process which, similarly to the NFS client, allows remote access to the file system held by Vice servers. Additionally Venus process allows the client to work during temporally unavailable connection to the Vice servers.

Files are organized in units called volumes. Volume refers to a subtree of shared namespace. All administrative tasks, like setting a quota, are performed over volumes. Volume is a main unit of replication. It means that each of the Vice servers hold complete replicas of a volume.

Coda realizes the *transactional semantic* of concurrent writes. It assumes that transaction is equal to session. When a client opens a file he caches the whole file on his side. All operations that are performed on the local copy are transferred to the server copy when the client closes the file. The client may reuse the local copy of the file in the next session as long

as the local copy is valid. The server registers all clients that have downloaded the file (*callback promise*) and when the file is modified by one of the clients others are informed by the server that their local copy became invalid (*callback break),* and their callback promise is erased.

Thanks to replication, Coda is resistible to network partitioning. A replicated volume is stored on multiple serves called the *Volume Storage Group* (*VSG*). The client's *Accessible Volume Storage Group* (*AVSG*) consist of those serves that belong to VSG and are accessible by the client. If AVSG is empty the client is *disconnected.*

Coda uses the *Read-One, Write-All* (*ROWA*) strategy of holding the volume replicas consistency. Each of the read operations can be performed on any replica but each write operation must be applied to all replicas. It is simple in a situation when VSG is equal AVSG, but if the network is partitioned for example due to the network link failure, Coda allows its clients to perform operations only on servers from its AVSG group. This *optimistic replication strategy* may lead to a situation that two AVSG hold a different version of the same file. To recognize such a situation each server in VSG holds for each file *Coda version vector* (*CVV*) defined as:

If $CVV_i(f)[j] = k$ then server S_i knows that server S_j knows at least version k of file f.

As an example we consider the Coda system that which contains three Vice servers: S_1, S_2, S_3. Before the network partitioning version vector of file f held by all servers was [1,1,1]. Then the network was partitioned such that AVSG of client A consists of S_1 and S_2 and AVSG of client B contains only S_3. Both clients A and B modify file *f*. After such situation $CVV_1(f) = CVV_2(f) = [2,2,1]$ and $CVV_3(f) = [1,1,2]$. Conflicts in the file version after the network reintegration can be recognized by comparing CVVs from each server. Some conflicts can be resolved automatically, other requires intervention. For example, when a conflict was caused by the modification of the same part of the file.

2.5. Google File System

The *Google File System* (GFS) [13] is a scalable distributed file system designed to work on inexpensive commodity hardware. Because the hardware component failures are the norm rather than the exception it provides fault tolerance. It is optimized for multi-GB files although KB-sized files can be stored too. Because in most GFS applications, files are mostly mutated. By appending the new data this operation is implemented efficiently. For the same reason GFS is optimized to large streaming reads and small random reads.

GFS is used in such Google applications as *BigTable* database described in [5] and *MapReduce* framework for the programming model (with the same name, described in [7]) used to process and generate large datasets.

A GFS cluster consist of a single *master* and multiple *chunkservers*, which are accessed by multiple *clients.* The storage machines are counted in hundreds or even thousands. The number of client's machines is similar.

Files stored in GFS are split into fixed-sized *chunks*. The typical chunk size is 64MB. Each chunk is assigned an immutable and globally unique 64 bit *chunk handle*. Chunks are distributed among chunkservers, which store them as a plain a Linux file that may be smaller than 64 MB and can be extended to this size if needed. Chunks are replicated on multiple chunkservers – typically three.

The master stores all system metadata. It holds the namespace, chunk locations, mapping from files to chunks. The master at startup asks each chunkserver about its chunks so this information does not have to be stored permanently. The master is also responsible for chunk leases (described later) and periodically: the garbage collection of orphaned chunks, chunk rebalancing, chunk re-replication if the number of valid chunks is less than the minimal

allowed, and sending a *HeartBeat* message to the chunkservers to give them instruction and collect their state.

GFS does not provide POSIX API (Unix semantic). Instead it provides its own API with usual operations like create, delete, open, close, read and write. Additionally it provides record append and snapshot. Snapshot creates a copy of a file or directory tree, which is exactly the same as the source in the time of calling the operation. This operation is efficient and its execution does not block the file modification. The append does not guarantee that the new data is added in the place that was the end of the file when the operation was called, but it returns the file offset where record was added. GFS assures that the whole record (not interleaved with other concurrent appends) is written from that place.

GFS has a relaxed consistency model. Each region of the file can be in one of the states*: inconsistent, consistent, defined*. According to [13]: 'A file region is *consistent* if all clients will always see the same data, regardless of which replicas they read from. A region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety'. A region is inconsistent when it is not consistent.

A successful region mutation not interleaved with other clients' mutations of the same region leaves data defined. Each read operation after that mutation returns a value that has been written. In case of successful concurrent modifications a region is undefined but consistent. Unsuccessful modification leaves a region in an inconsistent state. The application must be resistible to existence of inconsistent parts of the file.

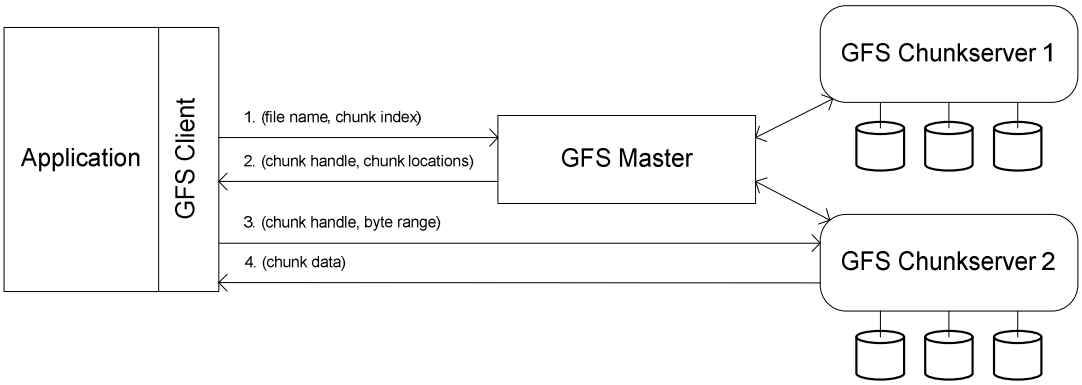


Fig 2.4. A Simple GFS read operation

Let us consider a simple read procedure is presented on Fig 2.4:

1. The client sends to the master a file name and a chunk index within the file. The client was able to calculate the index because the chunk size is fixed.
2. The master sends back a chunk handle and the locations of its replicas. The client caches this information.
3. The client contacts one of the replicas sending the chunk handle and a required byte range. Most likely it chooses the closest one. The distance is approximated according to the IP addresses.
4. The chunk server is response directly to the client.

Unless the file is reopened or information in the cache expires, the client does not have to communicate with the master for further reads of that chunk.

The GFS uses a mechanism of leases to choose a replica which is responsible for picking a serial order for all mutations to the chunk. This replica is called *primary*. A lease might be

extended if the initial time of 60 sec. was too short to finish mutation. The master may sometimes try to revoke a lease before its timeout.

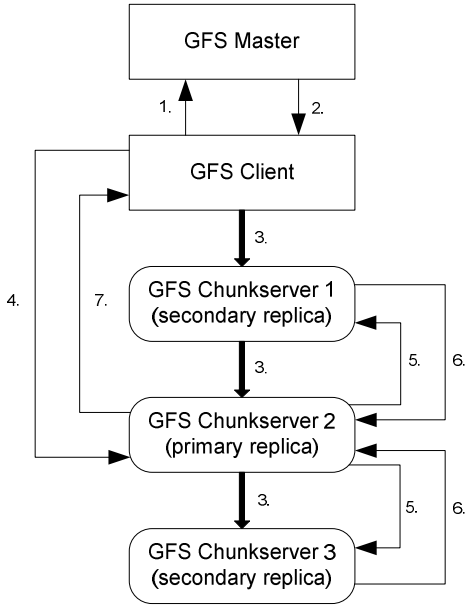


Fig 2.5. GFS write operation

Fig 2.5 illustrates the write process:

1. The client asks the master for the location of the primary and secondary replicas.
2. If none of the replicas has a lease, the master chooses a replica to be the primary. The master returns the requested information.
3. The client pushes the data to the one of the replica, which sends the same data to the closest replica which has not received data yet. The data is forwarded as soon as it is received by the replica. Such *pipelining* minimizes latency.
4. When all of the replicas receive data the client sends a write request to the primary. The primary organizes into sequence the write request, which has received from all the clients that want to modify the chunk. It applies the mutations in that sequence.
5. The primary sends to the other replicas a fixed sequence of chunk mutations.
6. After applying modifications in that sequence the secondary replica sends a confirmation to the primary.
7. The primary replies to the client. If no errors occurred the operation ends with success. Otherwise the write fails, leaving the replicas in an inconsistent state. The client code will retry the failed mutation.

The chunkservers work on a spread across many machine racks. The communication between the racks is slower than within the rack. The replica placement policy is used in three situations: chunk creation, re-replication and rebalancing. It focuses on three aspects:

- placing replicas on chunkservers with below-average disk space utilization,

- avoiding placing all new chunks on one chunk server to avoid high write traffic that is usually related with new chunk creation,
- spreading replicas of a chunk across racks.

Some of the GFS ideas were previously implemented in the xFS serverless network file system described in [1]. The Hadoop project is an open source framework inspired by GFS and MapReduce. Part of the Hadoop is similar to GFS *Hadoop Distributed File System* (*HDFS*). Further information about the project may be found at project website [38].

2.6. GFARM

Gfarm is an acronym for *Grid Datafarm*. It is a distributed file system created for the demands of the *Computational Grid*. Gfarm uses *Grid Security Infrastructure* for authentification. The first version of Gfarm, described in [31] was applied in high energy physic calculations. Gfarm v1 had some weaknesses: it was providing non POSIX-compliant *write-once* file semantic, and an application crash could cause metadata inconsistence. Gfarm v2 introduced in [33] solves those problems.

The main component of Gfarm are *Gfarm file system nodes* and *Gfarm metadata servers*. Apart from storage every file system node also provides computing resources. A job submitted to the Computational Grid will be executed on the node that has a copy of the requested file. This is so called *file-affinity* process scheduling. It is based on the observation that it is easier to *move the computation to data* than data to the computation. The Gfram metadata server is implemented over *OpenLDAP*.

Gfarm may be accessed through various ways: *Gfarm I/O library* as well as *scp, GridFTP and SMB protocols*.

The replication in the Gfarm file system is described widely in [32]. This article focuses on the performance of file replication. The *replica placement policy* is not described.

2.7. Other

A further description covers only selected parts of the presented systems. More detail about these systems may be found in related sources.

2.7.1. Wuala

Wuala is a a distributed storage system based on the *p2p network*. It was developed by Caleido AG – a company from Zurich, Switzerland. When Wuala was developed at the Swiss Federal Institute of Technology Zurich (ETH Zurich) it was known under the codename *Kangoo*. Some information about the product can be found at the product homepage [41]. Details of Wuala implementation were presented in Google Tech Talks [47] by founder of Caleido AG, Dominik Grolimund.

The Wuala provides the user with *persistent storage* that allows the user to store his own files in the p2p network. To make sure that data is always in the network Wuala uses *redundancy*. Instead of replication it uses *erasure codes*. Before being encoded, the file stored in Wuala is encrypted with *AES algorithm*. Then file is split into *m* parts and encoded into *n* fragments. Erasure codes have a useful property that allow them to reconstruct a file having any *m* fragments of *n*, and by this they allow greater file availability than in the case of replication.

When replication is used the availability of a file is expressed by the equation:

$$p_{rep} = 1 - (1 - p)^k \qquad (2.1)$$

Where:

- p_{rep} – the file availability in the case of replication
- P – the node availability
- k – replication factor

For erasure codes the same factor is expressed by the equation:

$$p_{ec} = \sum_{i=m}^{n} \binom{n}{i} p^i \left(1 - p^{n-i}\right) \quad (2.2)$$

- p_{rep} – the file availability in case of replication
- P – the node availability
- m – the number of parts that a file is divided into
- n – the number of fragments that the file chunks are encoded into

Let us assume a file that has 5 replicas and nodes that hold replicas are available for 25% of the time. The availability of the file is p_{rep} = 0 .763 in the case of replication. If erasure codes are used and the file is split into m = 100 parts and encoded into n = 517 fragments, approximately the same disk space is required (k = n/m = 5.17) , but the availability reaches p_{ec}=0.999.

2.7.2. Ficus

Ficus was created at the University of California Los Angeles (UCLA) as a distributed replicated file system for general use in nationwide networks. It is described in [15] and [27]. An interesting aspect of *Ficus* is so called *stackable layers* architecture shown on Fig 2.6.

The architecture of the system replication engine consists of two layers: *the logical and physical layer*. The logical layer abstracts multiple replicas and represents them as a single file. The physical layer holds file replicas. Both layers provide *vnode interface* – industry standard for mapping virtual file systems to Unix namespace. Because the physical layer provides *vnode interface* it allows the use of NFS as a transport layer to access a physical layer hosted on the different machines, so the *Ficus* logical layer is able to provide the user with access to the files from different hosts. Furthermore the *vnode interface* of the logical layer allows access to *Ficus* through the NFS from the other operating system. Therefore the client can connect to the *Ficus* file system remotely, and in this way access and modify all replicas of the file stored on multiple machines, staying connected only to one server.

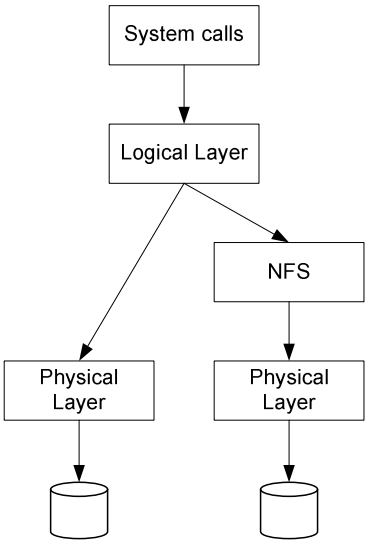


Fig 2.6. Ficus stackable layers

2.7.3. Content delivery networks

The content delivery network or the *content distributions network* (*CDN*) is a system of cooperating computers located across the Internet, which delivers content to the end users more effectively than the centralized server. The content mostly provided are large media files. The web content-based CDNs are Speedera, Sandpiper, Mirror Image and Skychache, Akamai and Digital Island.

These systems use replication or caching to achieve an increased transfer rate and reduced latency. The number of *replica placement algorithms* have been described and implemented. Some of these algorithms are compared in [19].

One of the possible replica placement algorithms is described in [28]. The article describes a *hosting service* that uses replication and relocation to increase overall system performance. The described algorithm requires knowledge about the network topology that is available in databases maintained by Internet routers.

2.8. Summary

The initial research in the field of the distributed system was focused on exploring the remote file access semantics and the related issues. As a result two protocols were developed - NFS and CIFS - and became the industrial standards for remote file access.

Later research focused on the replication (AFS, Coda and Ficus), which aimed to provide better performance and accessibility. However new concerns – assuring replicas consistency, the replica placement problem, toleration of hardware and communication failures (including network partitioning) - arose.

The most recent research does not tend to produce a system for general purpose. It is focused on specific applications (GFS – huge data sets sequential processing, GFARM – scientific applications in a grid environment). In these solutions, increase of performance is achieved at the cost of the relaxed semantic of the file access. In these systems the separation between data and meta-data processing is observed (the GFS Master is responsible for storing meta-data, the GFS Chunk server stores data, and the GFARM uses OpenLDAP as a meta-data server).

3. Theoretical aspects of distributed file systems

3.1. Interface

The client access to the file system is through the interface provided by the system. The operations and their meaning may vary in different systems. However there are some industrial standards of an I/O API. The *Virtual File System* (*VFS*) is one of them. It is the abstraction layer that allows uniform access to the various physical file systems, including distributed ones. If a distributed file system implements VFS we say that it implements *UNIX/POSIX I/O API*. However the implementation of this API can be computationally expensive, due to additional restraints put on the distributed file system. That is why some systems provide their own dedicated API. For example the GFS Client delivers to the applications an API to the Google File System. The API has regular *create*, *delete*, *open*, *close*, *read* and *write* methods, however their consistency guarantees are weaker than the corresponding operations in POSIX API. Additionally it provides *record append* and *snapshot* operations.

Another example of standardized API is *MPI-IO* [46] – the interface for parallel file access. It was developed by NASA and added to the parallel computing standard called *Message Passing Interface* ver.2 - *MPI-2*. It is designed to provide a file access to the system that utilizes a *message passing programming paradigm*.

The API that the system implements, significantly influences the system complexity and performance. The choice of API – suitable for the applications in which it will be used – is a key design decision and it is often a tradeoff.

3.2. Access model

There are two models of accessing files in distributed file systems. The first one is called *the remote access model*. In this model a client is provided with the transparent access to the files stored on the remote host. The client uses an interface that has the same methods as the interface of local files but the operations called by the client are executed by the server that contains the replica.

The second model is called *the download/upload model*. When such a model is deployed the client downloads the remote file to his own storage, then it utilizes the local copy. To commit changes it uploads its local copy back to the server.

3.3. Consistency models

The replication is a method utilized in many situations in the distributed systems. It is used for improving system performance or reliability. When there are many copies of the same data in a system there must exist a contract, which describes cooperation between the replicated resource and the processes that utilize this resource. This kind of policy is called *consistency model*. The term *a data store* comprehends systems like a distributed database, a shared memory, a distributed file system. A data store may be distributed among many machines. A consistency is usually defined in terms of read and write operations performed over the data store by parallel processes.

In his book, Tanenbaum [30] distinguishes two sorts of consistency models: *the data oriented* and *the client oriented*. The following sections are based on the taxonomy presented in the book. They include numerous examples, which utilize common notation to present details of consistency models. In the notation, several processes – denoted as P_n – perform read and write operations. The write operations are represented by $x \leftarrow a$, which means that the value a is written to the part of a data store marked x. Respectively read operations are denoted as $x \rightarrow a$, which means that reading part x returns a value a. The parts of data initially store have the value $\emptyset$. The timeline of subsequent operation should be read from left

to right. Some examples require additional symbols. They are presented in corresponding sections.

3.3.1. Data-centric consistency models

3.3.1.1. Strict consistency

A *strict consistency*, the most restrained model, is given by the following definition:

> *Any read to a memory location x returns the value stored by the most recent write operation to x.*

Since this model is implemented by the memory of uniprocessors, it is natural and obvious for every programmer. When some variable is read we expect the value of the latest variable update, not for example, the next-to-the-last. Because the model implies the existence of absolute global time it is practically impossible to implement in any distributed system. If there is more than one computer involved, there is always a time gap between the data request and delivery. During that period the read data might be modified. Assuming that all events are ordered according to global time, the strict consistency does not hold, because the modification operation was performed earlier and the read will return the outdated value. This situation is presented in Fig 3.1.b). In contrast Fig 3.1.a) presents an example of strict consistency, where process P_1updates x with value a and following the read operation of the process P_2 returns a.

Fig 3.1. Strict consistency a) holds; b) does not holds

3.3.1.2. Sequential consistency

The *sequential consistency* was introduced by Lamport in [21]. The model is defined as follows:

> *The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appears in this sequence in the order specified by its program.*

In other words any interleaving of operations is valid, providing that write operations are seen by all processes in the same order and the operations from a single process are performed in the sequence determined by its program.

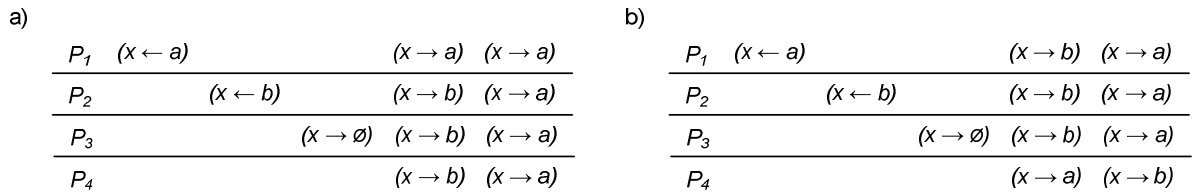


Fig 3.2. Sequential consistency a) holds; b) does not holds

The Fig 3.2.a) presents a data store, which implements the sequential consistency model. The processes observes that x at first is set to b, than to a, even though process P_1 executes write operation before process P_2. Similarly, process P_3 reads initial value – $\emptyset$, although processes P_1 and P_2 already have finished their writes. In the example Fig 3.2.b) the sequential consistency does not hold. The first inconsistency occurs in process P_4. Initially it

reads a then b, despite the fact that write operations appear in the opposite order in all other processes. Although process P_1 observes writes in that order – first b then a, its reads operations do not preserve sequential consistency. They violate the sequence of operations determined by the process program. Reading b after writing a, which is the last update of x, is equal to performing read first, then writing a.

The model is less restrained than strict consistency. As we can see, its definition does not refer to absolute global time, therefore it can be implemented in a distributed system or a multiprocessor unit.

3.3.1.3. Casual consistency

The *casual consistency* model proposed by Hutto and Ahamad in [16] relaxes the sequential consistency by separating the *concurrent* operations from the *potentially causally related*. In the general operation *A* is potentially causally related to operation *B* if the result of operation *A* may depend on the result of operation *B*. For example, if process P_1 sets a variable x and then process P_2 reads x and sets variable y it could possibly utilize the value of x to compute the value set to y. Two operations are concurrent if they are not potentially causally related. A data store is causally consistent, if it complies with the following rules:

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

a)

P_1	$(x \leftarrow a)$		$(x \leftarrow c)$		
P_2		$(x \rightarrow a)$	$(x \leftarrow b)$	$(x \rightarrow b)$	$(x \rightarrow c)$
P_3			$(x \rightarrow a)$	$(x \rightarrow c)$	$(x \rightarrow b)$
P_4			$(x \rightarrow c)$	$(x \rightarrow a)$	$(x \rightarrow b)$

b)

P_1	$(x \leftarrow a)$		$(x \leftarrow c)$		
P_2		$(x \rightarrow a)$	$(x \leftarrow b)$	$(x \rightarrow b)$	$(x \rightarrow c)$
P_3			$(x \rightarrow a)$	$(x \rightarrow c)$	$(x \rightarrow b)$
P_4			$(x \rightarrow c)$	$(x \rightarrow b)$	$(x \rightarrow a)$

Fig 3.3. Casual consistency a) holds; b) does not holds

A data store, which implements casual consistency, is presented in the Fig 3.3.a). Process P_1 performs two operations – first writes a then c. Process P_2 reads a from x and then writes b. Since value b might have been calculated using the read value a (but did not have to), the write operation performed by process P_2 is potentially causally related to the first write from process P_1. Therefore, in all process value a must be seen before value b. Because the second write of process P_1 is not potentially causally related to any other write operation, it is observed at a different moment in each process. The sequence of operations presented in the Fig 3.3.b) is not casually consistent. In this example process P_4 reads value b before a, and this violates the causality of operations.

3.3.1.4. FIFO consistency

The *FIFO consistency* (aka. *PRAM consistency*) was described by Lipton and Sandberg in [23] as a consistency model for Pipelined RAM. A data store implements the model if it fulfills a following condition:

> *Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.*

While constraints of casual consistency bounds the writes operations form the different processes, constraints of FIFO consistency refers only to the operations from a single process. The example in Fig 3.4.a) presents the sequence of operations in a FIFO consistent data store. As we can see, the writes from the same process are observed in the order they were executed. In other words, modification form process P_1 can be visible in any place – at the beginning, in

the middle or at the end – in the sequence of updates from process P_2. The example in Fig 3.4.b) does not hold FIFO consistency, because process P_4 observes writes from process P_2 in reverse order.

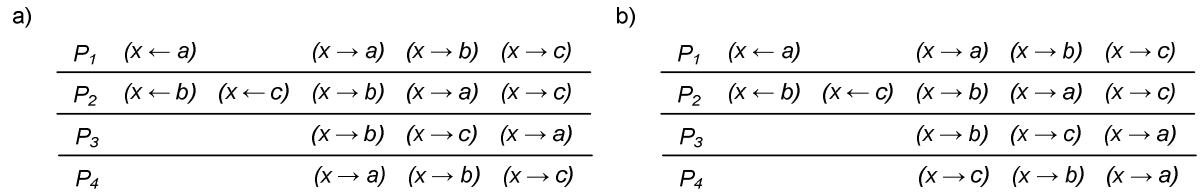


Fig 3.4. FIFO consistency a) holds; b) does not holds

The FIFO consistency is easy to implement in the distributed systems. Each process posses a local copy of a data store. The modification done by one process is broadcasted to the other processes. Each process maintains its own counter and numerates the subsequent update messages with increasing numbers. The processes apply the updates to the local copy of a data store according to the numbers assigned to the messages. Since the messages from each process have their own numeration, the processes have to store the number of the last applied update from each process. If a difference between the stored value and the number in the message is greater than one, the process postpones the updates from the message sender, until it receives missing ones. In the same time updates from other processes can be applied.

3.3.1.5. Weak consistency

In the *weak consistency* model – described by Dubois et al. in [10] –every process proceeds its operations on a local copy of a data store and does not have to pay attention to propagating the updates to the other copies. Especially, it does not have to take care of the order of updates. The model introduces so-called *synchronization variables* instead. The only operation on the variables is a synchronization ($sync(S)$). When it is finished, the changes made to the local copy of the data store are applied to the other replicas. It also fetches all updates from other copies. As a consequence, not every modification must be propagated during synchronization – only the most recent.

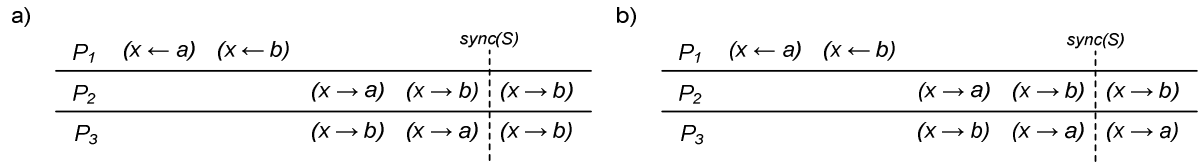


Fig 3.5. Weak consistency a) holds; b) does not holds

The definition of the consistency is generalized to more than one synchronization variable. A data store exhibits weak consistency if it respects the following rules:

1. *Accesses to synchronization variables are sequentially consistent.*
2. *No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*
3. *No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.*

As can be seen in the Fig 3.5.a) the data store, which is weakly consistent, does not give any guarantees about updates observed before synchronization. Processes P_2 and P_3 watch the updates from processes P_1 in different order. However, after synchronization, they both

observe the value of the last update. If they were observing different values, as it is showed in the Fig 3.5.b), the data store would not provide weak consistency.

3.4. Client-centric consistency models

In contrast to the data-centric models, which are useful for defining consistency for both distributed systems and a shared memory, the client-centric are mostly useful for the first group. The client-centric models describe the consistency from the perspective of a single process. The first model presented (eventual consistency) is the most lenient. It does not give any guarantees to single process. It was classified as client-centric, because it can be combined with the rest of client-centric models. The four other models come from system Bayou described in paper paper [34] and [35].

3.4.1.1. Eventual consistency

The *eventual consistency* is the weakest model presented here. In this model, assuming the temporary absence of updates, all copies of a data store converge toward an identical state. The resulting copies merge the state of all participants. A client is not given any guarantees about the version of data it observes, especially when it changes the replica it is utilizing.

The mirrored FTP server is an eventually consistent data store, which is used to accelerate file transfers on the Internet. Usually there is a main FTP server that all updates are performed on. The mirror servers are placed all over the network. They periodically compare their content with the main server. They create local copies of new files and delete obsolete ones. If the content of the main server was not changed long enough, all mirror servers are the exact copies of the main server. The client can utilize the nearest mirror server in the network to reduce latency, and as a consequence improve bandwidth of transfer.

3.4.1.2. Monotonic reads consistency

A data store provides *monotonic read consistency,* if it fulfills a following condition:

> *If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.*

The distributed World Wide Web server is a system that we would expect monotonic reads consistency. In such a system, web pages are replicated on the multiple physical machines. A client, which connects to the server, is redirected to the nearest server in the network or to the less loaded one. Suppose the client downloads a page, which contains the result of a currently pending football game. After a moment he refreshes it. His browser is redirected to a different server than the one that previously downloaded the page. The server may hold an older version of the page. The decreasing score is nothing that it expects, so the server should refresh its copy and send back a newer or at least the same version of the page.

3.4.1.3. Monotonic writes

A data store data store exhibits monotonic writes consistency, if it respects the following rule:

> *A write operation by a process on a data item x is completed before any successive write operation on x by the same process.*

The monotonic writes consistency could be sufficient for a distributed document management system. This kind of system allows users to work over the same documents. A document may exist in many replicas in the different servers hosting the service. Before modifying the document, the user must acquire a lock on it to avoid concurrent modifications. The document can be now modified. Suppose a user works on one replica and modifies it. Then he switches to another replica. He performs another modification and then the previous

update arrives to the replica. If the monotonic writes consistency was not assured, the previous modification could overwrite the new one. The model is similar to the FIFO consistency. However the definition of FIFO consistency refers to the entire data store in contrast to monotonic writes consistency, which is defined from the perspective of the process.

3.4.1.4. Read-your-writes consistency

A data store provides *read-your-writes consistency,* if it fulfills the following condition:

> *The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.*

A distributed user directory is an example of a system that should provide read-your-writes consistency. It provides authorization service for other systems. To improve efficiency of the solution the user record is stored in the multiple copies of the directory. The replicas in different parts of the network render their service to the local subsystems. If a user changes a password, he expects that after the operation successfully finishes, his password will be valid for all subsystems, which utilize the directory. In other words, no matter where he will use the password, it will grant him access. However, many real life applications implement only eventual consistency. In those systems a user password is updated in all directory replicas after some time. It leads to a situation in which, after a password change, the user might not be able to access a system, which utilizes a not actualized directory replica.

3.4.1.5. Writes-follow-reads consistency

A data store provides *monotonic read consistency,* if it has following property:

> *A write operation by a process on a data item x following a previous read operation on x by the same process, is guaranteed to take place on the same or a more recent value of x that was read.*

The distributed chat could be an example of a system that requires writes-follow-reads consistency. In the system, the distributed server stores the recent history of the conversation. The chat client connects to one of them and periodically reads the recent statements from the conversation. When a user writes something on a chat, his message is sent to the server that its client is utilizing. Then it is propagated to other servers. A user statement might be an answer for a question that he read from the downloaded chat history. If the writes-follow-reads consistency was not assured, the users utilizing other servers could see the answer before the question, due to differences in updates propagation delays. The described consistency model assures that, before the message will be saved in the server history, all other messages that the client read will be there too.

3.5. The reliable group communication

Before we present concepts related to the reliable multicasting we must explain some notions: *receiving* and *delivering*. We say that a message was received if it reached the desired node. In contrast we say that the message was delivered if it was received and passed to the application that had been waiting for that message.

Intuitively the term *reliable group communication* means that the messages are delivered to every process of the group. However, what happens if a process joins or leaves the group or the member crashes during the communication? Does the reliable multicast have to provide any message ordering? The following sections try to answer those questions and define the term reliable group communication more precisely.

3.5.1. The reliable multicast

The reliable point-to-point communication can be easily assured in the transport layer of any network. The situation with the multicasting is different. Networks often do not provide efficient multicasting or even if they do it is unreliable. For example on the Internet it is usually impossible to multicast message (using IP multicast) to the computers outside the local network. Furthermore, IP multicast does not provide any guarantees about the message delivery. The algorithms presented in this section work on top of the unreliable multicast layer. They ensure that all messages will be delivered to every group member. They assume that every group member is working and it is accessible through the network.

3.5.1.1. ACK

To ensure that messages will be delivered to every group member, the most obvious solution is to send back *acknowledgement* (*ACK*), any time the process receives the message. The sender retransmits the message to the host, which has not sent the ACK message. The confirmation messages are correlated with the messages according to a unique message identifier – a number from a sequence. The solution is inefficient, because if the group counts *n* members, the sender receives *n* ACK messages in response to every message he sent. As a consequence a sender is flooded with confirmations.

3.5.1.2. NACK

NACK (*negative acknowledgement)* [30] is a simple protocol for assuring a reliability, which overcomes the ACK problem. It can utilize any unreliable multicast protocol for spreading messages. Each message is tagged with a number from a sequence. When a new message is received, it is easy to check if any message was lost. If such a situation was detected the *retransmission request* is unicasted to the sender. Retransmission is sent directly to the receiver. Periodically the member sends a *session message* with a *sender id* and the sequence number of the recently received message. If it is necessary the sender retransmits all missed messages. The session message prevents the receiver from losing messages that were sent before long idle periods. It also allows the sender to periodically purge his message cache.

The NACK suffers a similar problem to the ACK. When the great number of recipients do not receive the message – for example due to router failure – the sender can be flooded with the retransmission requests.

3.5.1.3. SRM

SRM [12] is similar to NACK. It also can utilize any unreliable multicast protocol for spreading messages and the message sequence tag to discover lost messages. But instead of unicasting a retransmission request, the receiver multicasts it. Any member, which received the missing message, may answer to his request. The answer is sent through multicast so that any member, which also was missing that message, will not have to request retransmission. To limit unnecessary multicasting, members wait for a random period before sending retransmission. If they receive the retransmission they skip sending one by their own. Just like in NACK members in SRM regularly send the *session messages*.

The protocol effectively reduces the number of retransmission request and as a consequence, reduces the risk of NACK flood.

3.5.2. The message ordering

The multicast may order messages, but does not have to. There are three basic kinds of reliable message multicast: the unordered multicast, the FIFO-ordered multicast, and the causally ordered multicast. Additionally those protocols may be totally ordered. This means that all group members see the same sequence of the messages. It gives six different kinds of

ordering that reliable multicast may guarantee. The types of ordering are explained in further detail.

3.5.2.1. Unordered multicasts

A reliable unordered multicast does not give any guarantees about the message ordering. For example if there are three nodes that belong to a group: N_1, N_2, N_3 and node N_1 multicasts messages in order: m_1,m_2 then the messages might be delivered to node N_2 in order: m_2, m_1, but the messages delivered to N_3 might be in order: m_1, m_2.

3.5.2.2. FIFO-ordered multicasts

A reliable FIFO-ordered multicast guarantees that messages from one process will come in the same order as they were sent. As an example we consider a group of four members N_1, N_2, N_3, N_4. Node N_1 sends messages in order: m_1, m_2. Node N_2 sends messages in order: m_3, m_4. The messages might be delivered to N_3 in order m_3, m_1, m_4, m_2 and to N_4 in order m_3, m_4, m_1, m_2.

The FIFO-ordering is provided by the NACK protocol.

3.5.2.3. Causally-ordered multicasts

A reliable causally ordered multicast delivers messages in the order that preserves some causality. For example if message m_1 sent by node N_1 casually precedes message m_2 sent by node N_2 then m_1 will always be delivered before m_2.

The implementation of a protocol utilizes *vector timestamps* proposed by Lamport in [20].

3.5.2.4. Totally-ordered multicasts

A total-ordered reliable multicast guarantees that all group members will receive the messages in the same order. The sequence of message delivery may be any kind of the above, but it guarantees that all group members will receive messages in the same order. For example a reliable totally ordered FIFO multicast would deliver messages to N_3 and N_4 equally either in order m_3, m_4, m_1, m_2 or m_3, m_1, m_4, m_2 or in any other which fits to the FIFO ordering.

The total ordering can be implemented in various ways. The obvious and not very efficient idea is to send every message to the coordinator which multicasts it to all group members. If the communication provides FIFO, ordering all process will receive messages in the same order. The other idea is to utilize *logical clocks* proposed by Lamport in [22].

3.5.3. Virtual Synchrony

The protocols mentioned above do not consider failures of the group members. In the presence of process crashes, the multicast is expected to guarantee message delivery either to all non-faulty group members or to none of them. This property is also known as the *atomicity*. To ensure atomicity, the communication protocol should implement the *virtual synchrony* model introduced by Birnam and Joseph in [3] and [4].

The virtual synchrony is defined in terms of changes in the process group membership. Each process stores the identical list of members, the so-called *group view*. In the model every message is uniquely associated with a certain group view. It should be delivered to all processes from the view. The changes in the group membership, especially process failures, lead to the *view change*. The new view cannot be *installed*, as long as every message associated with the previous view, is delivered to all non-corrupted processes or to neither of them. In principle, the message can be ignored by all members only when its sender crashed. When the crashed member comes back, it is not in the group any longer. It must once again join the group. During that process, it can fetch the current state from any member of the group.

3.6. Summary

There are three architectural aspects that define the quality of service provided by a distributed file system – an interface, an access model and a consistency model. The interface defines a set of commands that a client uses to communicate with the system. If the file system implements the UNIX/POSIX I/O API the client perceives the system content as a local file system. It is usually convenient for a programmer, but ensuring the transparency causes a performance overhead. It is a reason for creating the dedicated interface, which has operations optimized for the purposes of certain applications.

The election of an access model is strongly related with the implemented consistency model. The download/upload model usually implies the implementation of the weaker consistency model.

The system interface is a contract between the system and its clients. Similarly the consistency model is an agreement between the clients, which defines how their operations coexist in the system. The efficiency issues of the consistency model are similar to the issues related to the interface. The more strict (and as a consequence convenient) the model is, the bigger the performance overhead it entails in a distributed system. It is an effect of a reliable group communication that must be involved in the system implementation. For example sequential consistency requires an expensive, reliable, total ordered multicast. Furthermore, the unreliable multicast protocols provided by the transport layers of the networks, require employing retransmission protocols like NACK or SRM. It has a negative impact on efficiency, just like the virtual synchrony model, imposed by the presence of process failures.

The general observation is that the admissible compromises on the consistency and the system guarantees improve the efficiency of a solution.

4. Replica placement algorithms

4.1. Problem definition

The replica placement algorithms (*RPA*) in the case of distributed files systems are responsible for solving *a file assignment problem.* It was defined in [9] as an optimization problem with constraints and has been proved in [11] that the problem is NP-complete. As a consequence it is solved using heuristics and the approximation algorithms. The numerous RPAs have already been proposed. An extensive review of the existing studies has been presented in [18]. The authors have presented a framework for evaluating replica placement algorithms and use it to compare different approaches to replica placement. Its part – t*he problem definition framework* – presented below is a unification of various problem definitions presented by authors of RPAs.

4.1.1. System Model

Formally stated the system consist of a set of clients *C*, nodes *N*, objects *K* and links *L*. Each physical link between two nodes from *N* is mapped with link in *L*. There are two variables defined:

- y_{ijk} – binary variable which indicates whether client $i \in C$ sends its requests for object $k \in K$ to node $j \in N$.
- x_{jk}– binary variable whether node $j \in N$ stores object $k \in K$

Using this variables four constraints are defined:

$$\sum_{j \in N} y_{ijk} = 1 \qquad \forall_{i,k} \qquad (4.1)$$

$$y_{ijk} \leq x_{jk} \qquad \forall_{i,j,k} \qquad (4.2)$$

$$x_{jk} \in \{0,1\} \qquad \forall_{j,k} \qquad (4.3)$$

$$y_{ijk} \in \{0,1\} \qquad \forall_{i,j,k} \qquad (4.4)$$

Equation (1) states that the client is accessing an object utilizing exactly one node. Constraint (2) indicates that only nodes that store the object can serve access to it. Constraints (3) and (4) state that objects cannot be split and requests to it are served entirely by one node.

4.1.2. Cost function

The existing RPAs define the cost function of a replica placement problem in various ways. The problem definition framework identifies parameters that are used, combined with variables mentioned above, to construct the cost function:

- $reads_{ik}$ – the read access rate to an object *k* by a client *i*.
- $write_{ik}$ – the write access rate to an object *k* by a client *i*.
- $dist_{ij}$ – the distance between a client *i* and node *j*. The distance could be measured with link latency or with any other link cost metric.
- mst_{jk} – the cost of update propagation initiated in node *j* and broadcasted to all other nodes which holds an object replica, expressed as a minimum spanning tree distance between these nodes.
- sc_{jk} – the storage cost of holding object *k* at node *j*. In this case, as the metric can be taken an object size, throughput of the node, or even a financial cost of storing a file at a specific node.
- $size_k$ – the size of object k in bytes.

- $acctime_{jk}$ – a timestamp of last operation over the object *k* at node *j*.
- hr_{ij} – a hit ratio of any cache on the path from client *i* to node *j*.

To clarify ambiguities let us consider the following examples:

- $\sum_{i \in C} \sum_{j \in N} reads_{ik} \cdot dist_{ij} \cdot y_{ijk}$ – the cost function defined in such a way refers to only a single object and read operations. The cost function does not consider a coexistence of many objects and their impact on the system performance. The RPA must be run separately for each one of them.
- $\sum_{i \in C} \sum_{j \in N} \sum_{k \in K} \left(sc_{jk} \cdot x_{jk} + reads_{ik} \cdot dist_{ij} \cdot y_{ijk} \right)$ – this cost function is defined for all objects. Additionally it includes a storage cost, however it only considers the influence of the read operations.
- $\sum_{i \in C} \sum_{j \in N} \sum_{k \in K} \left(reads_{ik} \cdot dist_{ij} \cdot y_{ijk} + write_{ik} \cdot \left(dist_{ij} + mst_{jk} \right) \cdot y_{ijk} \right)$ – this cost function includes influence of the update propagation. It is defined for all objects.

4.1.3. Additional Constraints

The problem definition might be extended with the additional constraints, to express additional features:

- *Storage Capacity* – the constraint on the maximum storage capacity of node.

$$\sum_{k \in K} size_k \cdot x_{jk} \leq SC_j \qquad \forall_{\mathrm{j}} \qquad (4.5)$$

- *Load Capacity* – the maximum request rate a node can serve.

$$\sum_{i \in C} \sum_{k \in K} (reads_{ik} + writes_k) \cdot y_{ijk} \leq LC_j \qquad \forall_{i,k} \qquad (4.6)$$

- *Node Bandwidth Capacity* – an upper bound on the number of bytes the node can transmit.

$$\sum_{i \in C} \sum_{k \in K} (reads_{ik} + writes_k) \cdot size_k \cdot y_{ijk} \leq BW_j \qquad \forall_{i,k} \qquad (4.7)$$

- *Link Capacity* – a constraint on the bandwidth of the link between two nodes. y'_{lik} is a binary variable which indicates whether client *i* utilize link *l* to access object *k*.

$$\sum_{i \in C} \sum_{k \in K} (reads_{ik} + writes_k) \cdot size_k \cdot y_{ijk} \leq CL_l \qquad \forall_{l} \qquad (4.8)$$

- *Number of Replicas* – a constraint on the number of replicas.

$$\sum_{j \in N} x_{jk} \leq P \qquad \forall_{k} \qquad (4.9)$$

- *Origin copy* (*OC*) – specifies the node which hold the original copy of an object.

$$x_{jk} = 1 \; for \; given \; j \; and \; k \qquad (4.10)$$

- *Delay* (*D*) – mathematically defined in [24]. A constraint, which specifies the desired maximum request time for a request in the system.
- *Availability* (*AV*) – mathematically defined in [24] and [28]. A constraint, which specifies the desired minimum availability of object in the system.

4.2. Existing algorithms

There were numerous RPAs proposed. We present a few as examples of different approaches for approximating the solution of the FAP. In the following descriptions, whenever it is possible we use the problem definition framework instead of original notation proposed by the authors.

4.2.1. Object replication strategies in Content Delivery Networks

In the model presented in [17] each Internet *Autonomous System* (*AS*) is a distinct node with a limited storage capacity. A goal of the presented optimization problem is to minimize the number of ASs that the user's request must traverse to reach the CDN server, which possesses a replica of an object. In other words the goal is to reduce the client perceived latency. Only read operations are considered. In terms of the problem definition framework the cost function is given by an equation:

$$\sum_{i \in C} \sum_{j \in N} reads_{ik} \cdot dist_{ij} \cdot y_{ijk} \qquad (4.11)$$

The storage limit is represented with constraint (4.5). The authors propose four heuristics to find an approximate solution. Two of them utilize the origin copy (*OC*) constraint given by variable x_0. It is a matrix similar to the previously defined x_{jk} having 1 when node *j* stores an origin copy of object *k* and 0 otherwise.

4.2.1.1. Random

The replica's location is established randomly, according to the following algorithm. The object is selected with uniform probability, in the same way the node is picked. If the node already contains an object the other node is selected.

4.2.1.2. Popularity

Each stores the objects that are the most popular among its clients. The node storage is populated with objects in the decreasing order of popularity until there is no space left. This operation can be performed by each node apart from the others. The entire knowledge required by this process node can be gathered by collecting the statistics of the request of its clients.

4.2.1.3. Greedy-Single

Each node *i* calculates the following function for each object *j*:

$$C_{jk} = p_k \cdot d_{jk}(x_0) \qquad (4.12)$$

Where:

- p_k – the probability that client will request the object *k*
- $d_{jk}(x_0)$ – the number of hops from the node *j* to the origin location of object *k*

Then the node sorts the objects in the decreasing order of results of the function (4.12) and fills its storage with the object according to that order until the storage space constraint is reached. In other words the node is fetching the most distant objects so that there is the greatest chance that client will request them. Just like in the previous example the node can perform this operation separate from the other servers. However it requires knowledge about the network structure and the origin of the objects to calculate $d_{jk}(x_0)$.

4.2.1.4. Greedy-Global

This heuristic is similar to the Greedy-Single, but it is computed globally. The CDN computes the value of the following function for all nodes and objects.

$$C'_{jk} = \lambda_j \cdot p_k \cdot d_{jk}(x_0) \qquad (4.13)$$

Where:

- λ_j – the aggregate rate of all request to the node *j*
- p_k – the probability that client will request the object *k*
- $d_{jk}(x_0)$ – the number of hops from the node *j* to the origin location of object *k*

It picks the node and the object, which have the greatest value of C'_{jk}. It places a replica in that node. The CDN runs the same procedure for the new placement denoted as x_1. The algorithm is iterated generating subsequent placements x_1, x_2, x_3,... until all nodes are filled. The algorithm evaluate an impact of a new replica to the entire system by recalculating (4.13) for all objects and nodes in each iteration.

The authors of strategies experimentally proved that Greedy-Global heuristic gives the best performance results, outperforming other strategies up to 24%. However test data that the inquiry was performed has some drawbacks – the p_k parameter is given globally so it does not consider distribution of interest in the objects in the network. For some nodes this parameter should be lower, and for others bigger. As a consequence the Popularity heuristic ends up with the same order of objects in all nodes limited only by a storage capacity. This is probably the reason why in [19], where the strategies were compared utilizing the real network traffic, based on WorldCup98 web logs [2], the Popularity heuristic was shown to be equally good as the Greedy-Global.

4.2.2. An Adaptive Data Replication Algorithm – ADR

The Adaptive Data Replication algorithm (*ADR*) [36] is an example of an algorithm that works for a tree network. This tree might be a physical network structure or a logical graph that reflects the physical communication links. The authors describe the algorithm with the following statement:

Metaphorically, the replication scheme of the algorithm forms a variable-size amoeba that stays connected at all times, and constantly moves towards the center of read-write.

The aforementioned scheme broadens when the number of read operation increase, and on the contrary it shrinks when the number of read operation increase. The scheme remains fixed as long as the numbers of read and write operations are equal, because it is optimal for the read-write pattern in the network.

The read operations are performed on the closest copy, while the write update is propagated along the edges of a minimal subtree, which contains writer and all nodes that holds the object. The cost function minimized by the algorithm, in terms of the problem definition framework, is given by the following equation:

$$\sum_{i \in C} \sum_{j \in N} \left(reads_{ik} \cdot dist_{ij} \cdot y_{ijk} + write_{ik} \cdot \left(dist_{ij} + mst_{jk} \right) \cdot y_{ijk} \right) \qquad (4.14)$$

The replication scheme is always a set of nodes which constitute a connected subgraph of the tree network. To achieve this the initial scheme is such a set, denoted by *R*, and any changes are made only on the border of such a scheme.

To explain the algorithm precisely we must introduce two groups of nodes. We say that a node is a:

- $\bar{R} - neighbor$ when it belongs to *R* and have neighbor which does not belong to *R*
- $\bar{R} - fringe$ when it is a leaf of subtree given by the set *R*

The ADR algorithm consists of three tests: an *Expansion-Test*, *Contraction-Test,* and a *Switch-Test*, which are used to determine whether there is a need for changes. The tests are run periodically in the time intervals given by a parameter *t*. The expansion test is executed by each node which is $\bar{R} - neighbor$, similarly the contraction test is performed by $\bar{R} - fringe$ nodes. If node is both $\bar{R} - neighbor$ and $\bar{R} - fringe$ it executes expansion test first. If set R contains only one node, this node executes the expansion test first and then if the test failed it performs the switch test.

4.2.2.1. Expansion-Test

During the expansion test the node *i* compares two integers – *x* and *y* – for each neighbor *j* which is not in *R*. *x* refers to the number of reads requests that *i* received from *j* in the last interval. *y* is the number of write requests that *i* received from itself or any neighbor except *j* during last period. If x > y, then *i* replicates investigated object into *j*. The expansion test succeeds if the test condition is satisfied for at least one neighbor, otherwise the test fails.

4.2.2.2. Contraction-Test

Just like the expansion test, the contraction test compares two integers – *x* and *y*. But in this case *x* is the number of writes that *i* received from *j* during the last interval. On the contrary *y* is the number of reads that *i* received in the last period form itself or from the neighbors different than *j*. If *x>y* then *i* requests *j* for leave permission. It cannot exit unconditionally, since *i* and *j* may be the only processors containing a replica. It waits until it receives a response, when *j* also wants to exit the replication scheme the leaving node is selected arbitrarily (for example the one with the smaller identification number).

4.2.2.3. Switch-Test

In the switch test for each neighbor *n* the node compares two integers – *x* and *y*. x is the number of all requests that *i* received from *n* during the last period, and y is the number of all other requests received in the same interval. If x > y the copy is moved from the node *i* to the node *n*.

In the paper the algorithm is extended to general network topologies. The authors prove that this algorithm is convergent. This means that algorithm will change the replication scheme as long as it can improve performance. When it reaches the point where no further improvement can be done, it stabilizes. It was shown that this point would be reached regardless of the initial replication scheme that was applied.

4.3. Replica placement compared to caching

The biggest competitor of the replica placement algorithm is caching. The cache stores objects locally using some strategy. Commonly it is the *last recently used* (*LRU*) policy. In this strategy there is a fixed size of cache. If an object is accessed, the cache is inspected whether it contains an object. If the object is present in the cache, the operation on the object is performed on the local copy. When not, it is accessed from the remote localization and the object is placed in the cache. When there is no empty slot in the cache the last recently used object is overwritten.

The comparative study between the RPAs and caching is presented in [19]. The author considers the usefulness of RPA and caching in the CDN (see: 2.7.3). The conclusion of their

work showed that caching is as good as or even outperforms the best RPAs. However this statement is only valid for present CDNs. Once the CDNs will provide consistency, availability, performance, and reliability guarantees (demands of DFSs) the RPAs start to be useful.

4.4. Summary

There are several goals that may be achieved using the RPAs – the improvement of performance, the improvement in availability of objects in the presence of communication failures, and the utilization of free storage space etc. These goals can be expressed as a combinatorial optimization problem. The presented problem definition framework provides the unified model, variables and parameters to express the cost function and the constraints for a specific replica placement optimization. Since most of them are the variants of FAP, which is an NP-complete problem, the heuristic is necessary to provide an approximate solution for them. Numerous strategies were proposed. In this section we have presented few of them, although they illustrate the groups that RPAs can be divided to.

The first division line separates the RPAs that ignore the rate of the client requests from the ones that are client aware. The example of such an RPA is the random heuristic presented in 4.2.1.1. It places the object replicas in a random manner, achieving the improvement for the entire system including, parts that do not require this object. On the other hand, ADR moves the file replicas in the direction of nodes that are performing an operation over this file.

The second division criterion is the scope of knowledge that the algorithm requires. We can distinguish the algorithms that require global knowledge about the system such as the network structure, the distribution of the operation rates among nodes etc. The example of such heuristic is the Greedy-Global. Other heuristics require only local knowledge, for example the Popularity heuristics require only the knowledge about the rates of the file access. This knowledge can be gathered by each node by observing the local traffic. Similarly the node running the ADR protocol needs to know who is its neighbor and must count the number of read and write operations from them. It is often easier to distribute the algorithm that does not require global knowledge.

The third criterion is whether the algorithm optimizes the access to a single file or to all the files in the system. The ADR is an example of the algorithm, which optimizes the location of the replicas separately for each file. It ignores the impact of the file for the performance of the entire system. The file which is rarely read is treated the same (possibly has the same number of copies) as the file which operations generate the most of the system traffic. The Greedy-Global heuristic, as opposed to the ADR, take into account demand for a file, considering the more frequent files in the first order.

The last criterion is the *reads/writes optimization*. Since these operations have different costs of network usage – read is performed by unicast, write by multicast – it must be considered by RPA in systems where write operations are significant. There are RPAs for a system where most of operations are reads, however it was shown that in such a situation, it is better to use caching instead. The presented object replication strategies for CDN ignore different characteristics of read and write operations. The ADR differently, the read/writes optimization is the main feature of the algorithm.

The important parameter is the algorithm reaction time for the changes in distribution of operations and changes in the network structure. The global knowledge algorithms react rather slower than the distributed ones. It is so, because it must gather the knowledge about the entire system and it must process larger amount of data. On the other hand the distributed algorithms can execute an algorithm more often on the smaller amount of data, just as in ADR.

5. System description

5.1. Motivation

The initial research on the distributed file system (DFS) showed that it is impossible to create a DFS for general-purpose. A DFS must be optimized for a specific application, e.g. Google File System is optimized to work with the BigTable database.

In recent years the dynamic development of the grid computing, the Web 2.0 Internet applications, the international business solutions and other applications which require storing and processing data in the locations distributed across a worldwide network, revealed the need for creating a distributed file system that is optimized to support such systems.

These applications share the common file access characteristic. The files are used in many different and often distant locations. The operations over the file are performed mostly from the spot the file was created, however it must be also accessible for other places. The file access intensity changes dynamically in time. The reads operations prevail the write operations, although the files are changed relatively often and the current version of the file must be accessible from all spots without unnecessary delays. The file system must be easily extensible but the changes in the file system structure are not very dynamical like in the p2p networks.

For example in an international enterprise system which holds the enterprise documents, an annual report is mostly utilized by the accountants by whom it is being prepared. Yet it may be accessed by the manager in a headquarters located in a different part of the world than accountancy.

The existing distributed file system does not meet demands of aforementioned systems. The analysis of the existing systems revealed several desired properties of the required solution:

- In contrast to NFS[1], which allows access to the files only from a central server, the files should be stored in the multiple nodes located among the different locations.
- The files from the different nodes should be presented to a user as one namespace. On the contrary to MS DFS, which allows only binding the shared folders into different places of one namespace tree. All nodes of the solution should be able to hold a replica of the file from any part of the namespace.
- It should be possible to hold more than one copy of the same file in the system. In addition the file should be the unit that is replicated as opposed to MS DFS where the entire folders are replicated or to Coda, where a volume is the replication unit.
- In systems like the GFARM an increase in performance was achieved by moving the computations towards places where files are stored. Nevertheless it is not always possible. For example, in business applications, we cannot expect an employee who wants to work on some document to travel closer to the file. So in the desired DFS the files replicas should be duplicated and relocated to achieve better performance of the system.
- The replication should be done autonomously by the system and not by an administrator like it is in MS DFS or Coda. It should respond dynamically to the changes in the file access rate.

[1] Understood as a client-server implementation of NFS available in most of Unix-like operating systems, but not as a remote access model.

- The replication process should be transparent to a DFS user. The replication and relocation should not interrupt operations over the files. It is an improvement in comparison to systems like MS DFS where only closed files can be replicated.
- The system should provide POSIX interface to resources, so files can be perceived as regular files. The GFS achieved better performance by changing the semantics of the file access, however it complicates the development of software that uses this system.

The Self-organizing Distributed Files System (*SoDFS*), which is the topic of this thesis, was created to reach two goals:

- Create a distributed file system that has aforementioned properties.
- Use a phenomenon of the self-organization to decentralize the process of replication and relocation of the file replicas, and reduce client perceived latency. (See more in Section 6.)

5.2. Architecture

Separation of *the storage service* and *the meta-data service* is a common tendency in the modern distributed file systems. The storage service is responsible for storing files or their parts and making them accessible. Meta-data service is responsible for holding the file system namespace and storage and replica location. For example in Google File System GFS Chunk servers constituting *the storage service*, and GFS Master plays the role of meta-data service.

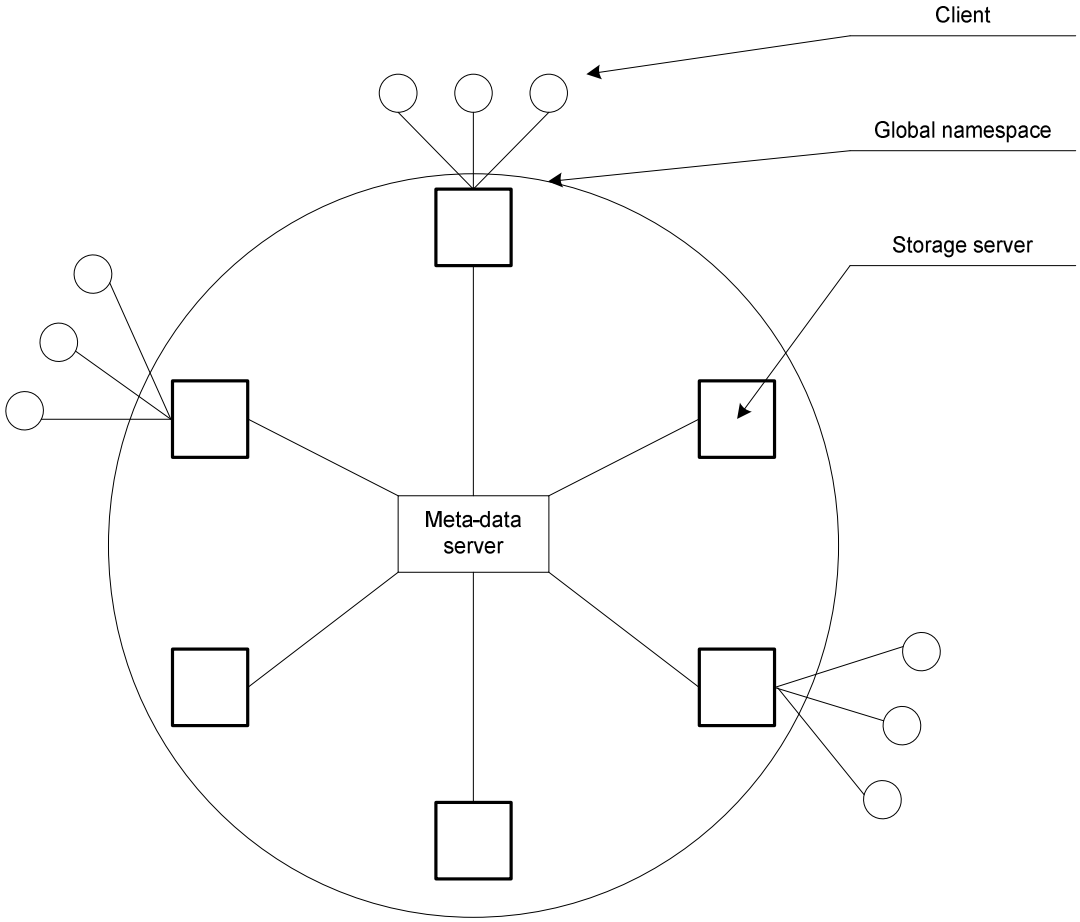


Fig 5.1. SoDFS architecture

The architecture of SoDFS is inspired partially by Google File System and partially by Ficus. The SoDFS consists of the s*torage servers.* Each storage server provides some storage space for the system. Simultaneously the storage servers are the gateways that allow the clients to connect to the file system. They present stored files in one namespace, which is held by a *meta-data server.*

When a storage server is registered for the first time in the meta-data server, it logs with a unique name. A meta-data server is assigned a unique *storage id*. Each file saved in the SoDFS is assigned a unique *file id.* A storage server can hold at most one replica of a file, so file replicas are identified by the pair (storage id, file id), similarly to chunks in GFS.

The files are stored in the storage servers using a file id instead of a regular file name. This approach allows complete separation of meta-data from data. For example the storage servers do not have to be informed about the changes in the directory or file name.

A client communicates with the file system only throughout one storage server – *the local storage server*. If the storage server does not contain a file replica, it is mediating between the client and *the remote storage server*, which holds a replica. This mediation was inspired by the Ficus stackable layers explained in Section 2.7.2. The operations over the file are performed by a client through the *file abstraction layer*. This part of the system is responsible for making multiple replicas to be visible to the client as a single copy (See more: Section 5.2.1). The remote replicas provide the same interface as the local replicas, but the calls are executed by the *Remote Method Invocation* (*RMI*) technology.

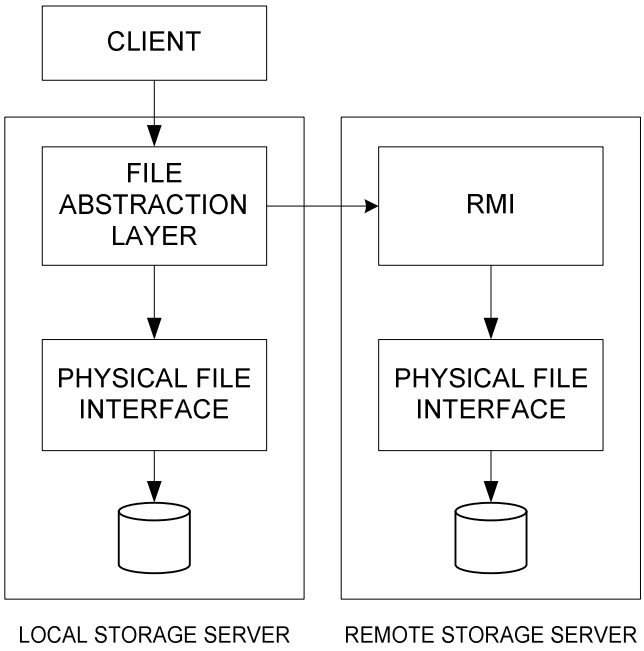


Fig 5.2. SoDFS file abstraction layer

When a client opens a file, the storage server, which it is connected to, sends a request containing a path to the file to a meta-data server. The meta-data servers send back a file id and some other meta-data of the requested resource. If the storage server does not contain the requested file, it asks for the list of the available replicas. The list is cached for some time if other clients, connected to the same storage server, would like to use the file. The storage server chooses the replica from the closest server in terms of link latency. The local storage server periodically checks the list of the available replicas and switches to the closest replica

if it is possible. If the replica has been moved or the storage server holding the file replica has failed, the storage server connects to the second best replica from the list.

The separate problem is removing files. If a file has been removed the storage servers are not informed, but only the meta-data record of the file is removed from the meta-data database. The storage servers periodically do the garbage collection of the orphaned replicas. If after the deletion of a file a new file with the same name is created, it will be assigned a different file id, so it will never happen that an old replica will be read.

When a storage server restarts after failure it requests the list of the replicas that it should have from the meta-data server, and gets the state of the replicas form the other storage servers.

5.2.1. Consistency model

The SoDFS is utilizes *read one, write all* (*ROWA*) policy of the file access. It means that the read operations are performed over one replica and write operations are performed over all replicas.

The file updates are sent to the storage servers holding the file replica by the group communication solution – JGroups. This tool implements the *virtual synchrony model* detailed in 0. When a client modifies a file, the update is sent to all replicas and the client is blocked. When the remote storage servers finish the operation it sends back a confirmation. The local storage server collects responses until it gets all responses or until it discovers that the server it is waiting for, is not working any longer. There are two methods of updates appliance ordering provided in SoDFS:

- the *total ordering* – all updates are applied in the same order to each replica, preserving that updates from the same storage server are applied in the order of generation.
- the FIFO – updates from the same storage server are applied in the order of generation. The updates coming from the different servers may overlap differently in each replica.

However even if the total ordering is used, it does not assure sequential consistency –the client can change the utilized replica between performing operations. As an example let us consider a file with two replicas. The client can read a file, when another client is performing an update. The client reads updated data from replica 1, and switch to replica 2, but an update was not applied to replica 2 yet, so next read can return old data.

As a consequence SoDFS is realizing the client oriented consistency model called: *read-your-writes consistency*. (Consistency models are explained in detail in Section 3.3.). This model guarantees that when the client has performed successfully a modification operation, the result of this operation will be visible for all the next read operations, unless it was overwritten by another modification operation from other clients. It can be assured, because the client is blocked.

5.2.2. Meta-data server

The meta-data server is responsible for several tasks:

- Holding a file system namespace – a hierarchy of files and directories.
- Replica location – providing storage servers with a list of storage servers which store replicas of the file.
- Storage server location – providing storage servers with current addresses of other storages servers.
- File locking – the meta-data server stores information about the locks that were set on the files.

An internal structure of the meta-data server, presented on Fig 5.3, is rather simple. It consists of three parts: a database, which assures meta-data persistence, the object-relational mapping layer and the meta-server interface.

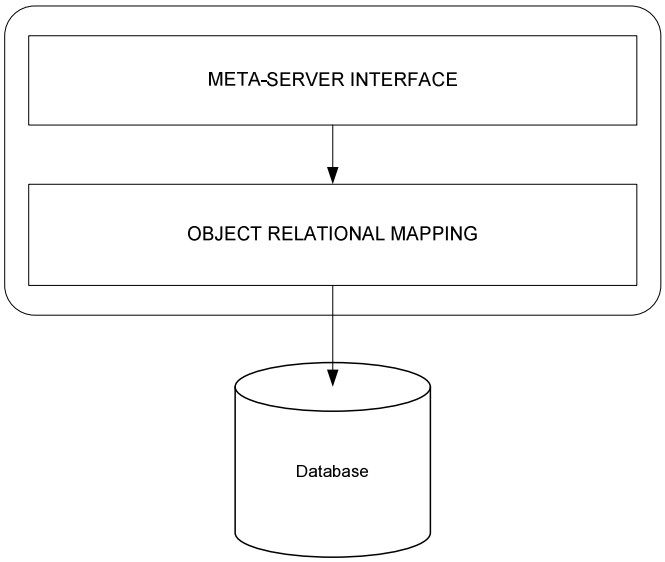


Fig 5.3. Meta-data server structure

The meta-server interface allows a storage server to access meta-data using *RMI* technology. The object-relational mapping is implemented with *Java Persistence API (JPA)* to perform operations on the meta-data database. The meta-data database schema model is presented in Fig 5.4.

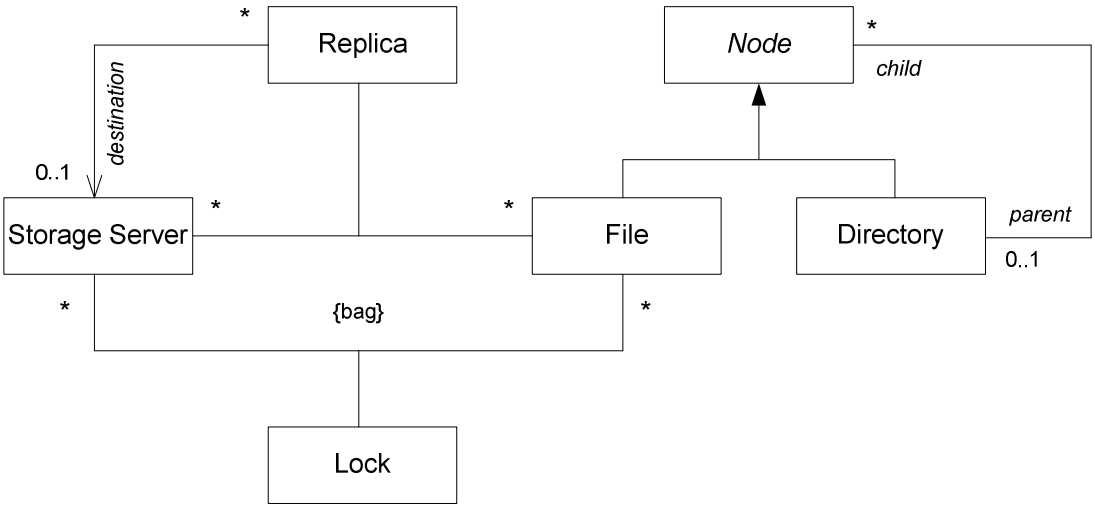


Fig 5.4. Meta-data persistence database schema model (UML class diagram)

5.2.3. Storage server

The storage server is responsible for holding the file replicas. It also constitutes a gateway for the clients to the file system. The storage server is provided as a driver to the third party solution Alfresco JLAN Server [42]. The product provides an API, that must be implemented, in order to make the file system accessible through the SMB/CIFS, NFS (POSIX interface) or FTP protocol. The storage server consists of several modules, presented in the Fig 5.5.

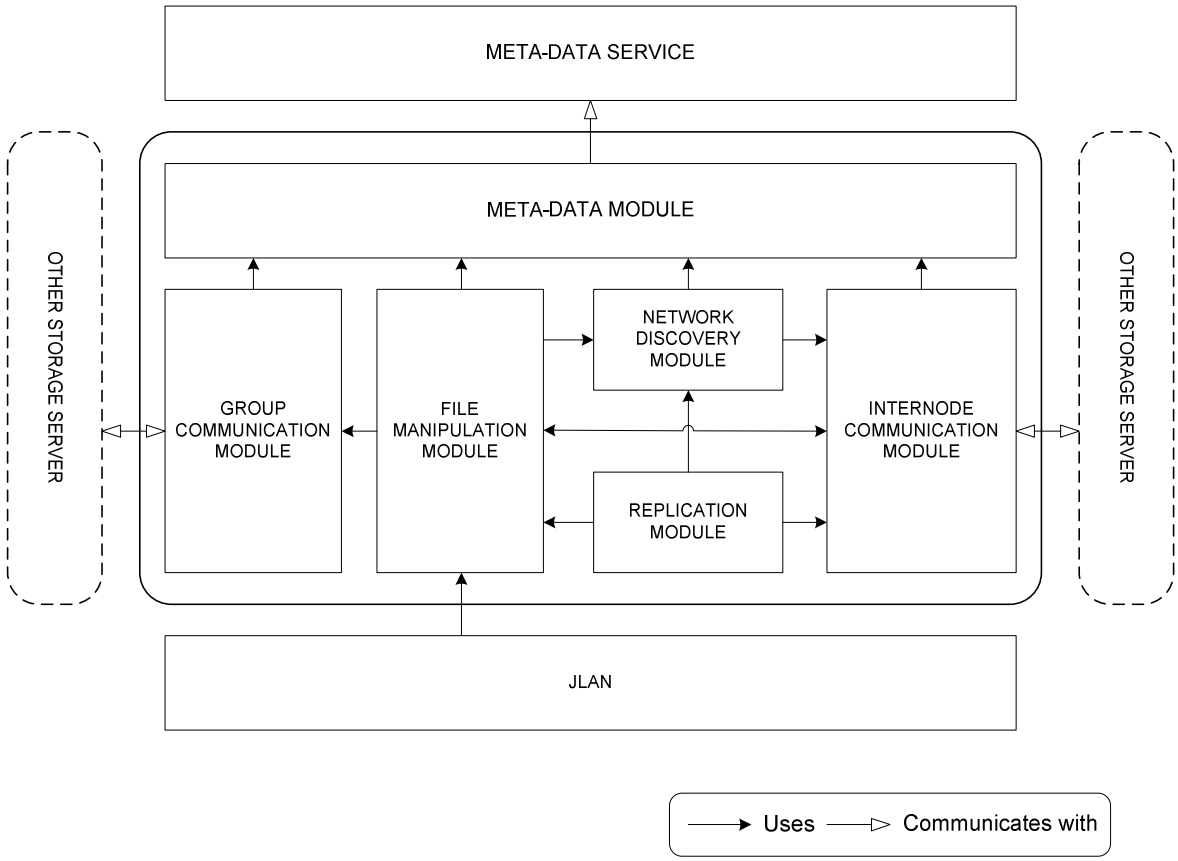


Fig 5.5. Storage server modules

5.2.3.1. Meta-data module

The meta-data module is an abstraction layer between the meta-data service and other modules of the storage server. It makes it possible to replace the meta-data server with another, possibly distributed, meta-data service implementation.

The meta-data module also contains a cache which reduces the number of requests to the meta-data service. The cache is fixed size and it uses *last recently used* (*LRU*) policy with an expiration time. It means that if there is no free space in the cache, the last recently used record will be overwritten. If the expiration time of the record passed it is removed from the cache.

5.2.3.2. File manipulation module

The file manipulation module is the central part of the storage server. This part of the module is the implementation of the JLAN virtual file system API.

The module is responsible for:

- registering the storage server in a system,
- mediating between JLAN and the SoDFS,
- providing access to the local replicas and remote replicas,
- performing operations over the file system namespace – file creation, deletion etc.,
- collecting the orphaned replicas,
- getting replicas of the replicated files, removing replicas of the dereplicated files,
- restoring the storage server state after failure.

The module uses:

- Meta-data module – for registering a storage server, performing operations on a namespace, registering and removing the file replicas information.
- Group communication module – for establishing communication groups for sending control data and updates to the file replicas.
- Network discovery module – for sorting the file replicas regarding the distance expressed in terms of latency.
- Internode communication module – for accessing the remote replicas.

5.2.3.3. Internode communication module

This module is responsible for establishing connections to other storage servers. The communication between the storage servers is performed using RMI. The module caches the RMI stub of the most commonly used storage servers. The cache uses LRU policy.

The module uses:

- Meta-data module – for getting addresses of the storage servers.
- File manipulation module – for providing replicas for the remote storage servers.

5.2.3.4. Group communication module

This module establishes group connections. It uses JGroups communication toolkit [44] which, implements the virtual synchrony model. The module creates two kinds of groups: *control groups*, and *update groups*. The control groups must assure total ordering of message delivery. The update groups provide the FIFO or total message ordering, depending on the passed configuration. The control group is used to exchange control communicates between the storage servers holding the replicas of a file. The update group disseminates updates among the storage servers.

The module uses:

- Meta-data module – for getting the list of the initial hosts of the group.

5.2.3.5. Network discovery service

The main task of the module is to evaluate the link quality to all other storage servers. This knowledge lets the storage server choose the best replica from the list. The round-trip time (*RTT*) was chosen as a metric for an evaluation of server proximity. This decision was done according to research described in [26]. The authors prove that this metric should be used when the client perceived latency must be reduced and this is the goal of the SoDFS.

In RTT the total time between sending a small message and receiving a response is measured. The response for the message is sent back immediately. The RTT is measured by taking an average from n trials (in SoDFS default is 3).

The SoDFS measures the *RTT* time to all other storage servers. This solution will not scale for the larger number of nodes, yet it is sufficient for the needs of evaluation.

The module uses:

- Meta-data module – for getting the list of the storage servers.
- Internode communication module – for measuring RTT to other storage servers.

This module is also used by the SoRPA algorithm described in Section 6. The algorithm requires the determination of *the closest neighbors*. Intuitively it is the group of storage servers that has the lowest *RTT* to the local storage server (the node which performs a network discovery).

To define the closest neighbors notion precisely we introduce several auxiliary terms:

- S – is a set of all storage servers except the local storage server
- $RTT: S \to \mathbb{N}$ – is a function which maps a storage server to the measured RTT
- S_n is a finite sequence of all elements from S sorted by RTT
- $\Delta(n) \triangleq RTT(S_n) - RTT(S_{n-1}), \quad n \geq 2$
- $\mathrm{cut(n)} \triangleq \frac{\Delta_{\max}}{a^{\left(\frac{n-2}{b}\right)}}, \quad n \geq 2$

where:

- $\Delta_{\max}$ is a maximal allowed difference between the first and second closest neighbour
- b and c are parameters which control *cut* function congruency

The set of the closest neighbors (*CN*) is a minimal set which matches two conditions:

$$s_1 \in CN \tag{5.1}$$

$$s_n \in CN \Leftrightarrow \left(s_{n-1} \in CN \wedge \Delta(n) < cut(n)\right) \tag{5.2}$$

In other words, the allowed changes in RTT for the initial members are bigger than for more distant. As an example let us consider the scenario presented in Tab 5.1.

n	*RTT(n)*	*cut(n)*	*Δ(n)*
1	10		
2	25	20	15
3	32	15,98127	7
4	41	12,77005	9
5	48	10,20408	7
6	60	8,153709	12
7	66	6,515332	6
8	69	5,206164	3
9	71	4,160056	2
10	72	3,324149	1
11	73	2,656206	1
12	74	2,122477	1
13	75	1,695994	1
14	76	1,355207	1
15	77	1,082897	1
16	78	0,865303	1
17	79	0,691432	1
18	80	0,552498	1
19	81	0,441481	1
20	82	0,352772	1

Tab 5.1. Closest neighbours determination. Example scenario.

The parameters values are Δ_{max} = 20 ms, a = 1.4, b = 1.5. In this example the first five storage servers were qualified to be the closest neighbors. The sixth was not accepted because its Δ(n) was bigger than the cut(n). The scenario is visualized in the chart presented in Fig 5.6.

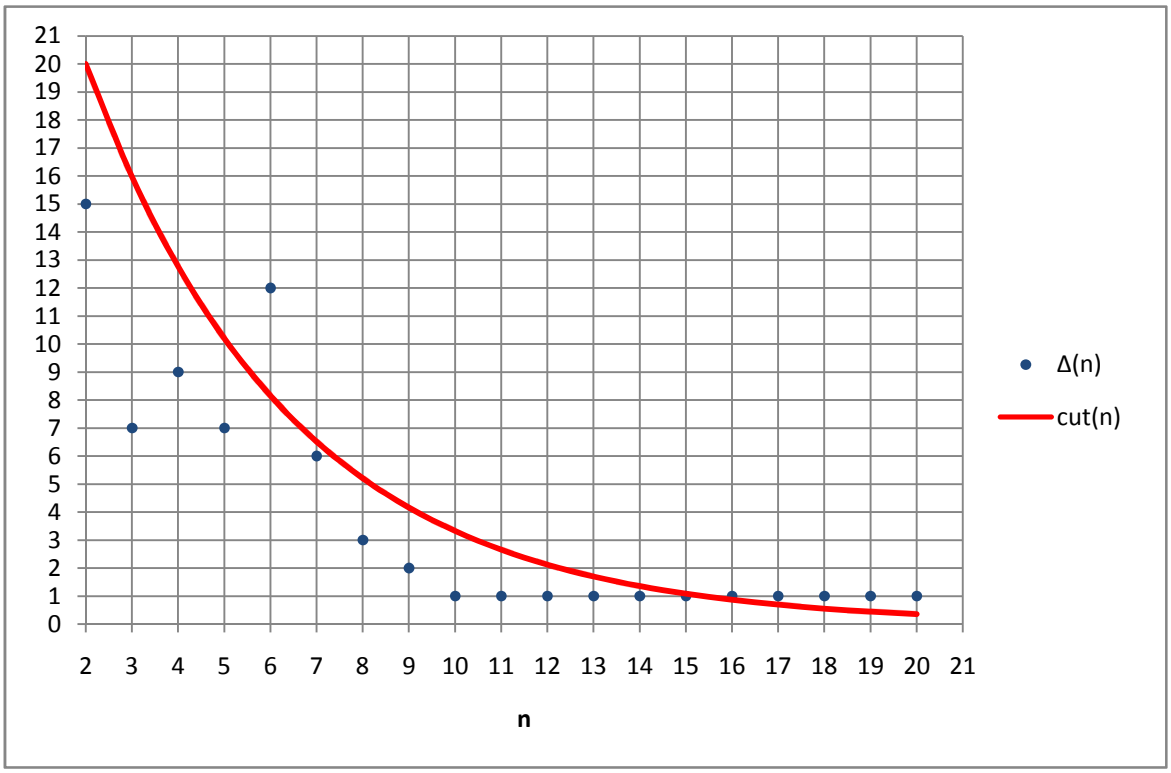


Fig 5.6. Closest neighbours determination. Example scenario visualisation.

5.2.3.6. Replication Module

The replication module is an implementation of the SoRPA algorithm explained in Section 6. It is responsible for running the replica placement algorithm, and sending and receiving replication requests and orders.

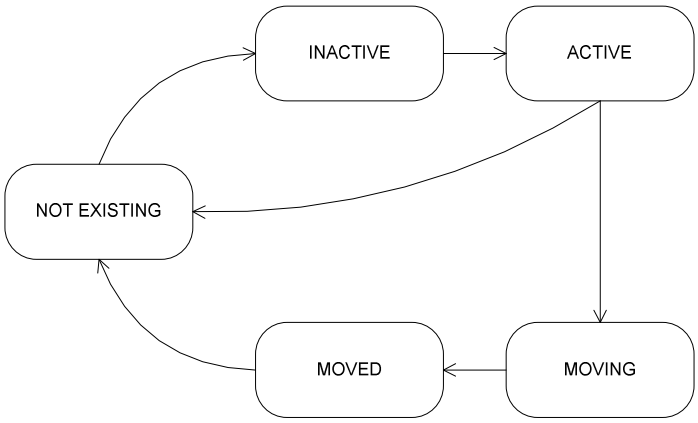


Fig 5.7. Replica status in the meta-data server (UML state diagram)

Because, the decision to remove a file replica - dereplicate the file – is made by each storage server autonomously it could be possible that all file replicas could be dereplicated. To avoid such a situation, a control communication group is created for each file. The group

delivers the messages in the same order to all members. If a storage server wants to possess the replica it is joining the group. It fetches the group state – the list of actual members. Then it sends a *join message*. When the group members receive the join message, they are adding the joining host to the list.

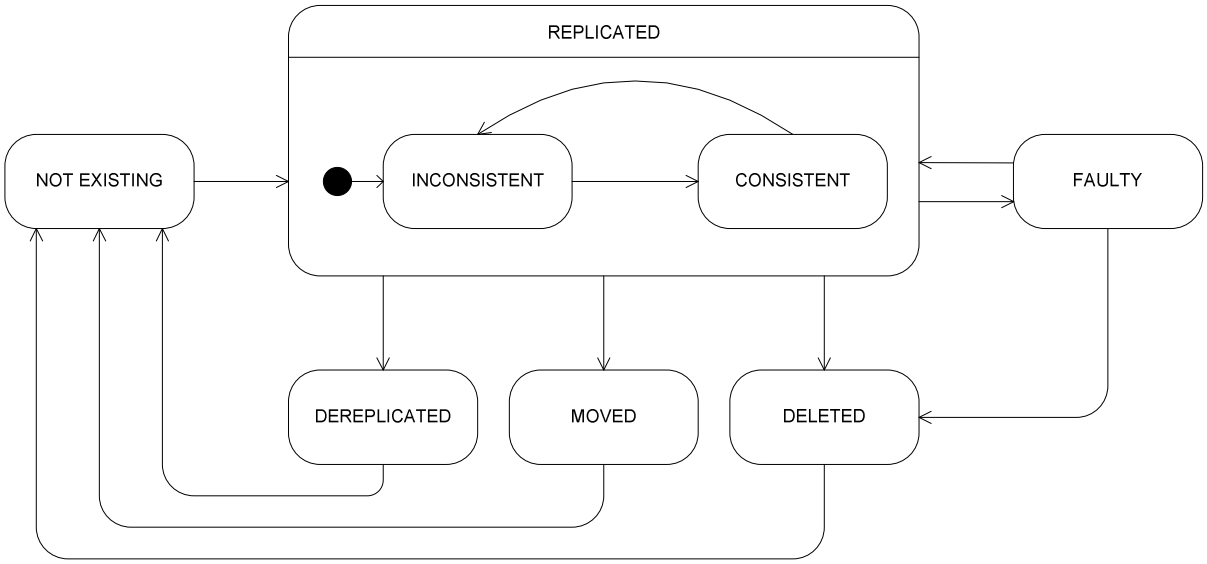


Fig 5.8. Replica state in the storage server (UML state diagram)

If the storage server wants to remove a replica it sends a *leave request*. The members remove the requesting node from the members list only if the list size is bigger than NOR_{min} parameter – the lowest number of replicas allowed. The node leaves the group only when it receives its own message and it is able to remove itself from the list. The join and leave messages come in the same order in each node, so each node can determine if the dereplication was successful or not.

Two different finite state machines describes the state of a replica in the system. To distinguish the state of the replica in the storage server from the state from the perspective of the meta-data server, the last one is called *a status*. The possible changes of the status are showed in the Fig 5.5. Initially the replica is *not existing.* The replica has the status *inactive* if it is being added to the storage server. After the replica is installed in the storage server, its status is changed to *active*. In case of dereplication the replica become *not existing* again. The other possibility is that the replica is *moving*. When a new replica is activated in the destination of the move, the moving replica become *moved* and waits for the garbage collection. The state diagram describing the state of replica in the storage server is presented in the Fig 5.6. Similarly to the status, initially the replica is *not existing*. When it becomes *replicated* it may be in sub states: *consistent* and *inconsistent*. The replica is in state *consistent* if it is connected to the update group of the corresponding file and the data of that file has been fetched (See Section 3.5.3). In the case of an *Input/Output errors*, like drive failures the replica is becoming *faulty*. It requires an administrative intervention to make back the replica *consistent.* When the replica is *dereplicated*, *moved* or the file is *deleted* the replica clients are informed about the state change and the replica become *not existing* (after garbage collection in case of the move and the deletion).

The module uses:

- File manipulation module – for performing replication and dereplication.
- Network discovery module – for selecting nodes for the coins exchange.

- Internode communication module – for coins exchange and receiving and sending the replication requests and orders.

5.3. Fault tolerance

When the distributed file is concerned three kinds of failures are possible:

- a server failure – power outage, hardware failure etc.,
- a communication link failure – the link to the network is interrupted,
- a network partitioning – the router failure may lead to a situation where two or more network partitions are constituted and the computers from the different partitions are not able to communicate with each other.

The fault tolerance was not the main concern of the work. The single meta-data server is a single point of failure. Although the system is not resistible for meta-data server failure, the SoDFS tolerates two first kinds of failures of the storage servers. The file system is obligated to maintain a minimal number of replicas (NOR_{min}) replicas of the file, so the system will work correctly even if $NOR_{min} - 1$ storage server was temporarily unavailable, due to the link or server failure. Furthermore the storage server is able to rebuild its state after restart utilizing information from the meta-data server.

5.4. Security

Extending the SoDFS to be a fully secure service is a task for a separate project. There are four areas that must be secured:

- communication between the storage server and the meta-data service,
- communication between the storage servers,
- communication between the client and the storage server,
- group communication between the storage servers.

The first two issues can be solved by establishing the RMI communication over SSL channel and by adding additional authentication.

The third issue can be solved by configuring the JLAN properly. Its CIFS implementation includes support for NTLMv1, NTLMv2, NTLMSSP, SPNEGO and Kerberos/AD authentication methods.

The group communication between the storage servers is realized by the JGroups, which provides its own built-in configurable security.

Furthermore if a SoDFS storage space were provided by an untrusted vendor, the data stored locally by each storage server should be encrypted. This situation may occur in a GRID environment, where different companies and institutions can share their storage space. It could lead to unwanted access to the locally stored data with omission of the SoDFS authentication.

5.5. Technology and tools summary

The SoDFS was fully implemented in Java. The system uses several third party tools, which simplify the implementation or even made it possible.

5.5.1. Alfresco JLAN Server

Alfresco JLAN Server [42] is an embedded virtual file system, which is presented to the end user as a shared drive. It delivers numerous useful services like an enterprise authentication support or multiple access protocols – CIFS, NFS, FTP. It is available on OEM or GNU public license.

5.5.2. JGroups

JGroups toolkit [44] provide reliable multicast communication, which implements a virtual synchrony model. It was successfully used in multiple products including JBoss – J2EE Application Server. JGroups is open source licensed under the Library (or Lesser) GNU Public License.

5.5.3. Oracle TopLink Essentials

Oracle TopLink Essentials [45] is an open source implementation of Java Persistence API (*JPA*) that is part of EJB 3.0 specification [39] published under Java Community Process℠ Program (JSR-000220). It is used in the project as an object-relation mapping tool in the meta-data server.

5.5.4. Apache Derby

Apache Derby [37] is an open source relation database utilized in the system for holding the meta -data.

5.6. Summary

The SoDFS has all the desired properties mentioned in Section 5.1, so the first goal of the work was achieved. The evaluation presented in Section 7. will show whether the second goal was accomplished.

The system is ready for an experimental deployment, however before running in a production environment it should be improved. The meta-data server, which is the single point of failure, should be exchanged with a distributed meta-data service in the future. The system also requires the aforementioned improvements in security and fault tolerance.

6. Self-organizing Replica Placement Algorithm

6.1. Notation

On the sketches presented in this section we use a simplified notation to represent the network structure. As we see in Fig 6.1 we skip presenting routers and instead we draw only links between the routers (represented as triangles), which are directly connected to the storage servers (circles) or are connected to more than two links. Although clients (squares) do not have to be connected to the storage server with a direct link, we assume that clients utilise the nearest storage server to interact with the file system. The system administrator decides, which storage server is the closest one. As a consequence we present all clients as directly connected to the storage server. In the following illustration the client marked red contacts its storage server through two routers, in the simplified sketch it is connected directly to that storage server.

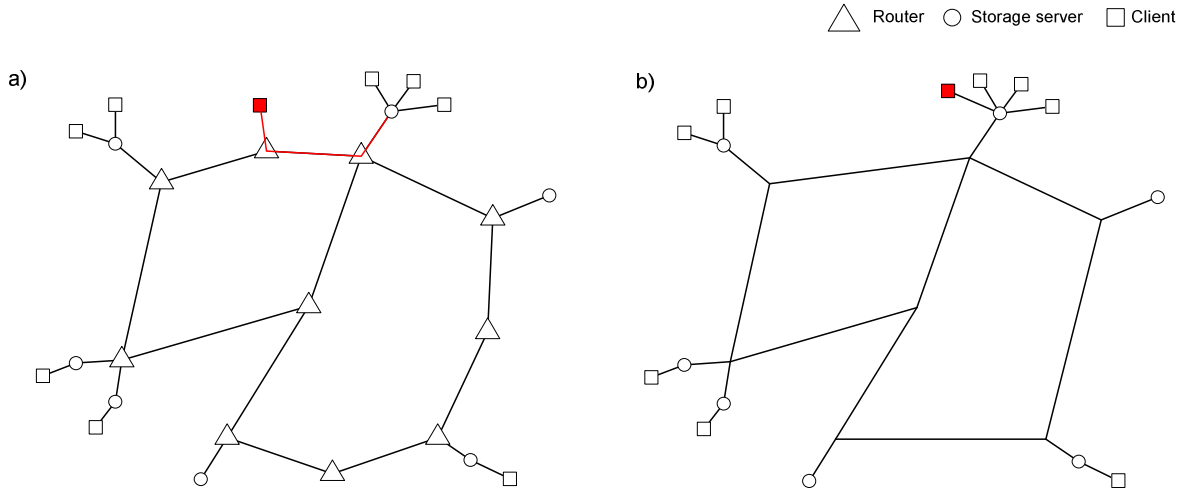


Fig 6.1. Simplified notation a) network structure b) simplified sketch

6.2. Problem definition

The following problem definition is presented according to the problem definition framework presented in Section 4.1. When studied carefully, the model of SoDFS does not match the system model introduced in the framework. The SoDFS client is incompatible with the client of the system model. All operations over the files are performed by the SoDFS client through a single storage server. Conversely the model client is accesses the files directly from the closest node, which holds a desired replica. To make both models consistent we must assume that the SoDFS nodes are also the clients in terms of the system model framework. Such defined clients generate traffic according to their internal processes – the SoDFS clients.

The cost function of problem we would like to solve is given by the following equation:

$$\sum_{i \in C} \sum_{j \in N} \sum_{k \in K} \left(reads_{ik} \cdot dist_{ij} \cdot y_{ijk} + write_{ik} \cdot \left(dist_{ij} + mst_{jk} \right) \cdot y_{ijk} \right) \qquad (6.1)$$

There is an additional constraint on the number of replicas:

$$\sum_{j \in N} x_{jk} \geq NOR_{min} \qquad (6.2)$$

The workload balancing, the disk drives performance and the utilization of free space among the storage servers should not be considered by the proposed algorithm.

6.3. Motivation

The reviewed replica placement algorithms showed that they are burdened with various weaknesses. The heuristics like *Greedy-Global* require a global knowledge about the system. To calculate replica placement it must be provided with the structure of a network connecting the peers, the file usage statistics from every node etc. This kind of information is not always accessible or gathering it might be expensive. The simple heuristics, like *Random*, improve the performance also in the parts that do not require it, because they omit information about the demand for certain files in a distinct part of the network. Others, like *Popularity* heuristic, do not take into account the difference between the cost of reads and writes. Since write operations involve update propagation to all replicas which have a cost larger than the cost of a read. The algorithms like (*ADR*) provide read/write optimization and utilize only local knowledge, however they work over some specific overlay structures, which entail an additional cost for building them.

The proposed algorithm should overcome these problems. It is designed for the requirements of a distributed file system. It should accomplished several goals:

1. It should be highly distributed.
2. It should scale up to work efficiently in a world-wide network.
3. It should dynamically change the location of replicas as a reaction to the changing in time, number and distribution of operations over the files.
4. It should compromise between the read and write optimization proportionally to the number of read and write operations.
5. It should improve performance of operations of the clients, which utilize a file not to the entire file system.
6. It should work only in the application layer.

6.4. Design issues

During the design of a heuristic for a self-organizing replica placement algorithm we must consider several design criteria. We discuss them in the following points.

6.4.1. Read/Write optimization

In most distributed file systems only one file replica is necessary to process a read operation, in contrast writes on one replica cause an update that must be propagated and applied to all replicas. As a result efficiency of read a operation depends on distance between the replica and client - more precisely between the storage server that holds the replica and the storage server the client is connected to. If we assume that all storage servers read their local data with the same speed, the best performance is achieved when the replica is placed on the same storage server that the client is utilizing.

For writing, the distance between a client and a storage server, which performs the operation is also significant, however the most important for performance are the number of replicas and distances between them. The greater the number of copies and distances between them, the longer update propagation time that is required.

Fig 6.2. visualizes the relation between reads and writes optimization. The green circles represent the storage servers, which hold replicas, the green squares represent clients that read the file. The green polygon shows the part of the network that will participate in the upgrade propagation. When the replicas are closer to the clients, read operations are performed faster, when replicas are closer to each other writes performance increases.

As we see, reads and writes optimization are at a variance, so the replica placement algorithm must make a trade-off between them.

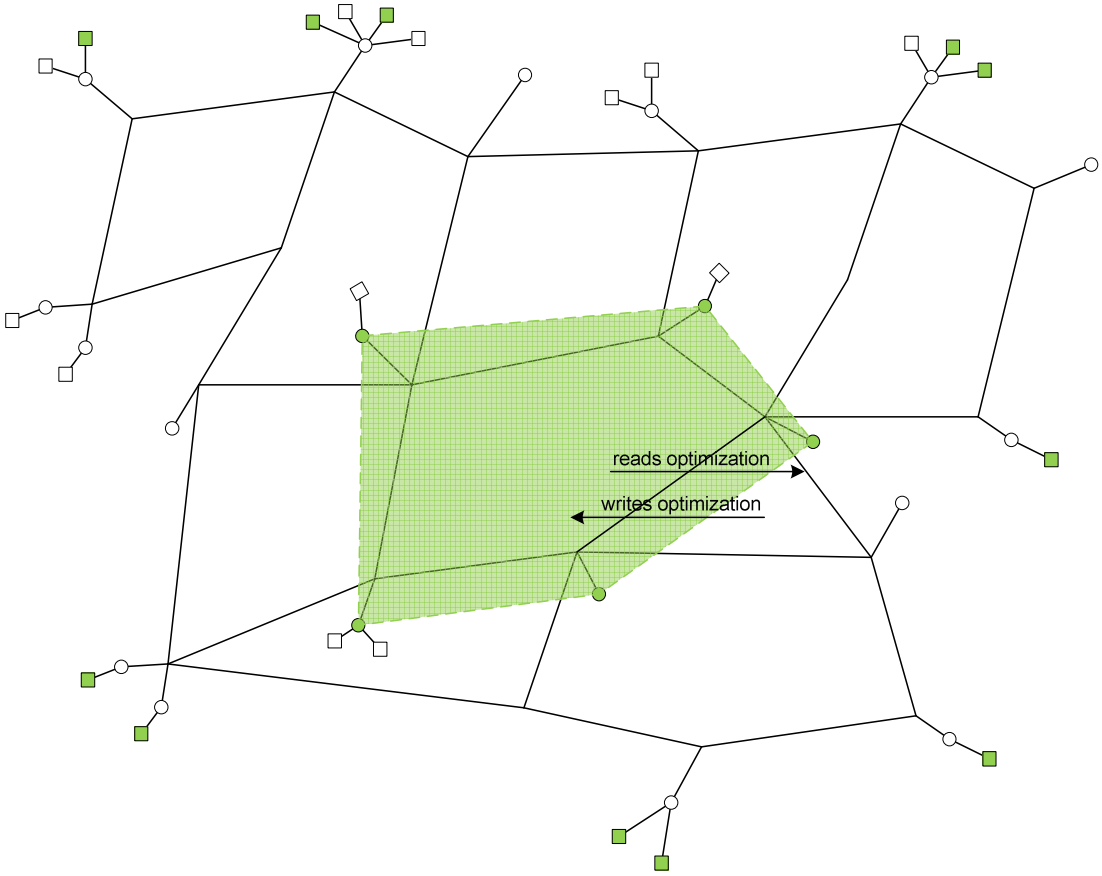


Fig 6.2. Read/Write optimization

6.4.2. Locality

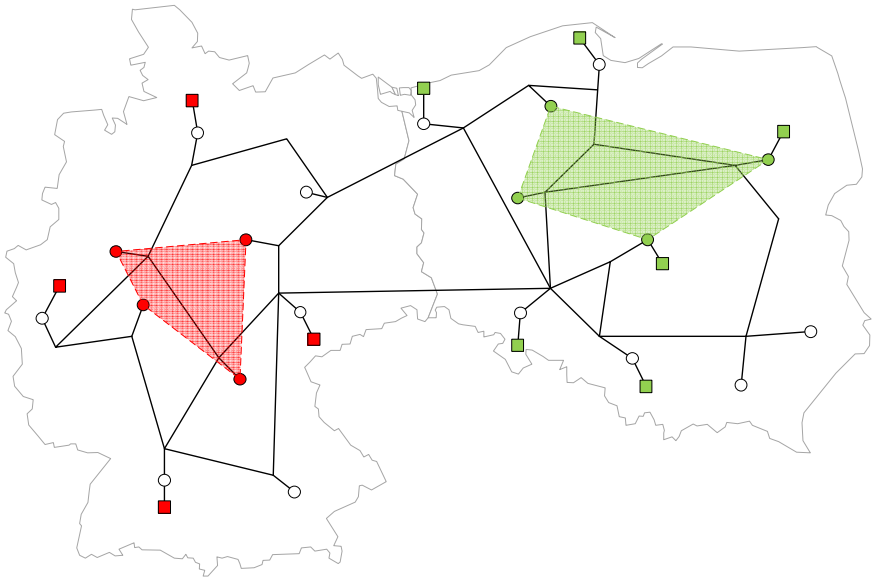


Fig 6.3. Locality

In a number of applications the files are used mostly in a certain geographical localization. For example, in the social networks pictures of one user are mostly viewed by his friends. Most of them live in the same city or region as the user. So most of the operation performed on those files comes from a certain part of network. The replica placement algorithm should take advantage of this property and propagate the replicas to that part of the network to improve performance for the clients that are localized in that part.

The example, presented in Fig 6.3., shows the location of two files i.e. video clips from a video sharing website. One clip is in the Polish language, the other is in the German language. The storage servers and the clients who perform the operations on the Polish clip are marked green, the German clip is marked respectively red. The clips are viewed mostly in the countries from which they come. The replicas of the videos are placed in the regions from which the files are accessed. This regions match the countries where the clips are mostly viewed.

6.4.3. Dynamicity

In the case of the distributed file system, a replica placement algorithm should adapt the location of the file replicas according to the changing distribution of the operations over the file, rather than creating one permanent placement of replicas at the moment of the file creation. As an example we may once again consider the video sharing website. For example a video presenting some prank at the beginning may be popular only in one country, but after some time it may become very popular in other countries or even worldwide. If replicas were placed permanently in the country that the file was added, after the interest explosion in other regions, the copies would be accessed from distant locations, causing latencies for the clients.

A replica placement algorithm should be reactive not only to the increase in the interest, but also to the reduction of interest. If the number of operations over the file decreased, the number of replicas should also be reduced. It is possible that some file would become popular in one part of the network, meanwhile interest in that file would decrease in the region where the file was previously popular. In such case dereplication in the part of reduced interest would be desired, while simultaneously new replicas should be created in the region of increased interest.

The replica placement algorithm for the distributed file system should also respond dynamically to the changes in a system structure. It should readjust the replica locations with reference to events like adding new links to the network, a temporal network link failure or its permanent removal, a temporal or permanent storage server failures and the addition or the removal of the storage servers.

6.4.4. Servers location and network structure

For replica placement algorithm knowledge about the storage servers locations and a network structure between them is crucial. A system which works in the application layer has no means to obtain precise knowledge about a network structure that it utilizes. Fig 6.4. presents such system, where two clients, connected to the storage servers labeled A and B, are reading the same file form the storage server C. The length of the edges is proportional to the latency of the link. The clients generate the same workload to the server C. There is a node D directly connected to the router, which routes packages from A and B to C, but D does not take part in routing! None of servers A,B or C can check that there is a node D close to the routing path between them, in case if they use only interface of transport layer (TCP, UDP). This node is the best location for moving the replica from C. If the replica was placed in node A it would decrease performance of the client connected to the node B, and vice versa. The node D is closer to both of the clients.

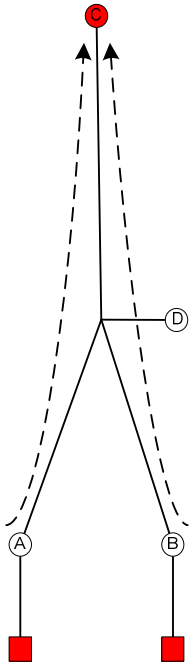


Fig 6.4. Local knowledge

This system could use a network layer to retrieve that information from the network routers, but using a lower network layer might be complicated. The structure of the network could be provided by a system administrator, yet for a system with thousands of nodes it could be a strenuous task to create and maintain the up-to-date network plan. It could approximate the network structure according to the local transfer speeds and latencies measured by each storage sever. However, calculating the network map on the basis of such measurements is also a complicated process. The decentralized solutions may use a local knowledge about the network structure. Each peer can use a local image of the network structure limited to the closest neighbors of the node. This domestic view is defined by the administrator or approximated according to measurements.

6.5. Algorithm

6.5.1. Inspirations

The proposed replica placement algorithm takes advantage of the self-organization phenomenon, which refers to the ability of a system to achieve an increase in the internal organization without being managed or led by any outside system.

It is inspired with two self-organizing algorithms – *the ant colony optimization algorithm* (*ACO*) and *the epidemical algorithms*. The first was proposed by Marco Dorigo in 1992 in his PhD thesis [8] as a solution for the problem of finding a path in the graph. It simulates ant behavior in finding tracks from a colony to the food. Initially the ants are traversing environments using the random paths. They leave a pheromone mark on their track. If an ant finds food it goes back to the colony using a trace left on the route. When the ant, which has found the path to food, comes back to the colony, the pheromone mark on their track becomes stronger than the mark on the track of ants, which have not found food. It is because the ant has traveled that path twice. The stronger the pheromone mark is the more likely it is that other ants will follow such a track. As a consequence the route to the food is more traveled. This causes a feedback loop, because the more ants wander on the path, the more pheromone marks are gathered there. The pheromone evaporates in time, so the unfrequented paths vanish. ACO simulates this process, letting virtual ants wander over a graph and leave the

pheromone over its edges. It can be used to produce close to optimal solutions to the traveling salesman problem.

The epidemical algorithms were developed in 1987 in Xerox Palo Alto Research Center [6] as a method of dissemination updates in distributed database. In a *rumour spreading algorithm,* one variant of epidemical algorithms, the replicas of the database are *gossiping* about the updates (rumors). When one copy of database was updated, it become *infective* (in terms of the epidemiology theory) other copies are *susceptible*. The replicas are continuously communicating at random with each other. When an infective node comes in contact with a susceptible one it passes the update to it. The infected replica become also infective and also propagates the update. After some time the infected node stops passing the update to the other nodes, it is removed. As a consequence all replicas are informed about the update.

6.5.2. Idea

In the system which implements the *self-organizing replica placement algorithm* (*SoRPA*) every storage server holds a *coin list* and performs three parallel activities:

a) a coins generation,
b) a coins exchange,
c) a coin list analysis and a replica management.

Every read or write operation performed by clients connected directly to the storage server adds with some probability a *coin* (*token*) to the coin list. The coin has a similar role to the pheromone in ACO. The coin list represents the distribution of demand for files of the clients connected to the storage server, which holds that list.

During the exchange a node sends a token to one of its randomly selected neighbors. The neighbor adds a token to its list and the node removes it from its own. In a similar manner to gossiping, the coin exchange completes information provided by the coin list with information about the file demand of the surrounding nodes. The token has some expiration time after which it is removed from the list. Granted that the operation distribution is constant, an equilibrium of coins is reached as a consequence of the combined process of the coins generation, exchanges and expiration.

According to its list, a node performs an analysis of which files it needs most and sends *a replica requests* to the nodes that already hold replicas. Each node gathers received replica requests and periodically process those requests making one of the possible choices: creating a replica by a requestor, moving a replica to a requestor, or ignoring a request. It also can dereplicate any of its replicas when they are no longer needed.

The example presented on the Fig 6.5. illustrates the basic aspects of the algorithm. The coins exchange activity will be explained in detail later so the example does not present the explanation. In the pictures we see a distributed file system containing seven storage servers. Four of them take part in reading two files, marked blue and yellow. A storage sever which holds a replica is marked with the file color, just like a client who uses that file. Step 1. presents the coin generation. The node A, which contains a replica of the blue file, has one connected client, which utilizes the local replica of the blue file. The operations of that client have generated three coins – marked blue, as they were generated due to manipulation over the blue file replica. The node B contains a replica of yellow file, but a client who is connected to that node utilizes the blue file using the replica in storage server A. So the coin list of that node contains only blue tokens. The node C does not hold any replicas. Its client reads the yellow file with a slightly lower rate than blue file clients, which were previously mentioned. As a consequence of its operations two yellow tokens were generated. The node D also does not have any replicas, but it has three clients. Two of them operate on the blue file, one on the yellow. Their manipulations generated one yellow and four blue tokens.

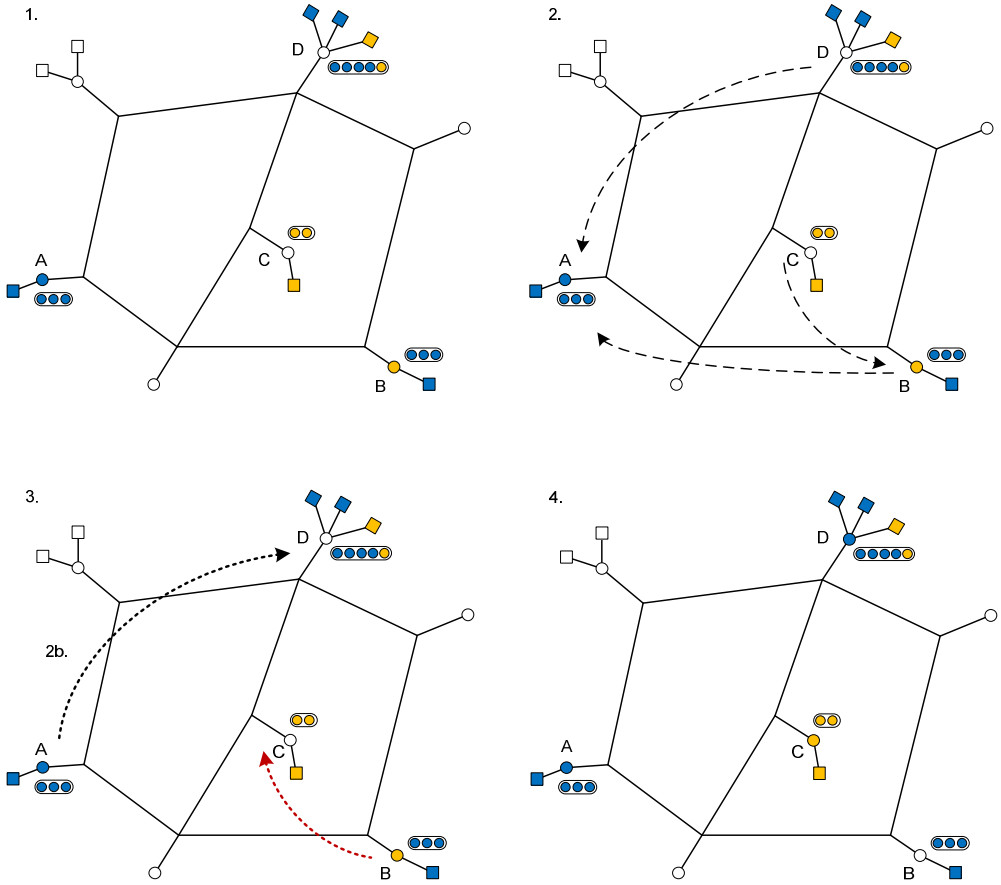


Fig 6.5. Basic flow

Step 2. presents the analysis of the coins list. According to the coins list in the node A, it can be calculated that for this node the most required is the blue file. Nevertheless this storage server already contains the replica of that file, so in this step the node takes no action. In the same way node B detects that it also needs a replica of the blue file, so it sends the replica request to the node A. Similarly, node C sends the replica request to the node B. Node D has a token in both colors, but it can determine from the proportion of coins that a blue file replica is more desired. It sends a request to the node A.

In step 3. nodes A and B are considering the replica requests. Node A received two replica requests, so it compares the number of blue coins in each requestor's coins list. Since node D has more blue coins than node A, it will receive a replica of the file. The node A also has a comparable amount of coins so it will keep the replica. On the contrary the node B has no yellow coins so it will move its replica to the requestor – the node C. As four activities of algorithm (here presented three) run continuously or in short periods we constantly improve a replica placement.

In the next few paragraphs we present in depth each of the algorithm activities.

6.5.2.1. Coins generation

Every file operation executed by the client directly connected to the storage server can possibly generate a coin, no matter if the file accessed is a local or remote file. However the file manipulations commissioned by the other storage servers in the name of their local clients do not generate tokens.

The *probability of coin creation,* p_c parameter, is defined globally for the entire file system. It approximately defines the percentage of operations, which will create a coin. Every token holds information about the operation by which it was created:

- id of a manipulated file,
- id of a storage server, which has generated a coin,
- id of a storage server which holds a replica of the manipulated file,
- operation type: read or write,
- amount of read or written data,
- the number of visited nodes (*NVN*),
- expiration time.

The *expiration time* is a property, which defines time after which a coin will be removed from the coin list. The nodes must have loosely synchronized clocks, because the tokens may have to be removed by the other storage server that it was generated (due to the coins exchange). The expiration time is set at the token creation. The value is a sum of the current time and *the coin time to live* (TTL_c), defined for each file. This parameter defines the period from which tokens are sufficient for analysis. If we have a file that is accessed very intensively, but for example, every day at a certain hour and the rest of the day there are no operations over this file, the TTL_c should be set to twenty-four hours. After the initial time equals the TTL_c, analysis may start at any time and find the tokens from the short period of intensive utilization. For the file that is used with a constant intensity the TTL_c might be shorter – an hour or even a minute.

6.5.2.2. Coins exchange

As already mentioned, since other storage servers do not take part in the routing of other nodes traffic, or are not aware of it, it is hard to detect that one node is between several others, which use the same file and that it is possibly a good place for holding a replica. To solve this issue we introduce coins exchange as a method for propagating local demands for the files to the surrounding nodes. As a consequence, in the node can be gathered a significant number of coins, which refers to the same file and comes from other nodes. It means that possibly a node is a convenient place for a replica. An example of this behavior can be found at Fig 6.6. This case was already mentioned in Section 6.4.4 - to access a file the clients connected to the nodes A and B utilize the node C, the node D is near the routing paths between other nodes. The node holding a replica is marked with red color.

At the beginning the coin lists nodes A and B are becoming filled with coins, due to operations performed by the clients. At the same time nodes are exchanging tokens. The nodes A and B passed four tokens to the node D. After the coins analysis the nodes send the replica requests to the node C. The node C moves the replica to the storage sever D, because it has the greatest number of coins and storage sever C itself has none. The storage server D is the best place for the replica, because if the replica was moved to the node A it would decrease performance of the node B client and vice versa.

The coins exchange should be done proportionally to the frequency of file operations executions. The probability of coin exchange, p_e parameter, is defined globally for the entire file system as a product $p_e = kp_c$ where k is a factor which defines the percentage of coins that should be exchanged.

In every exchange, one coin is pushed to the randomly selected neighbor. The selection of the neighbors is not part of SoRPA. The algorithm used in the SoDFS is explained in Section 5.2.3.5.

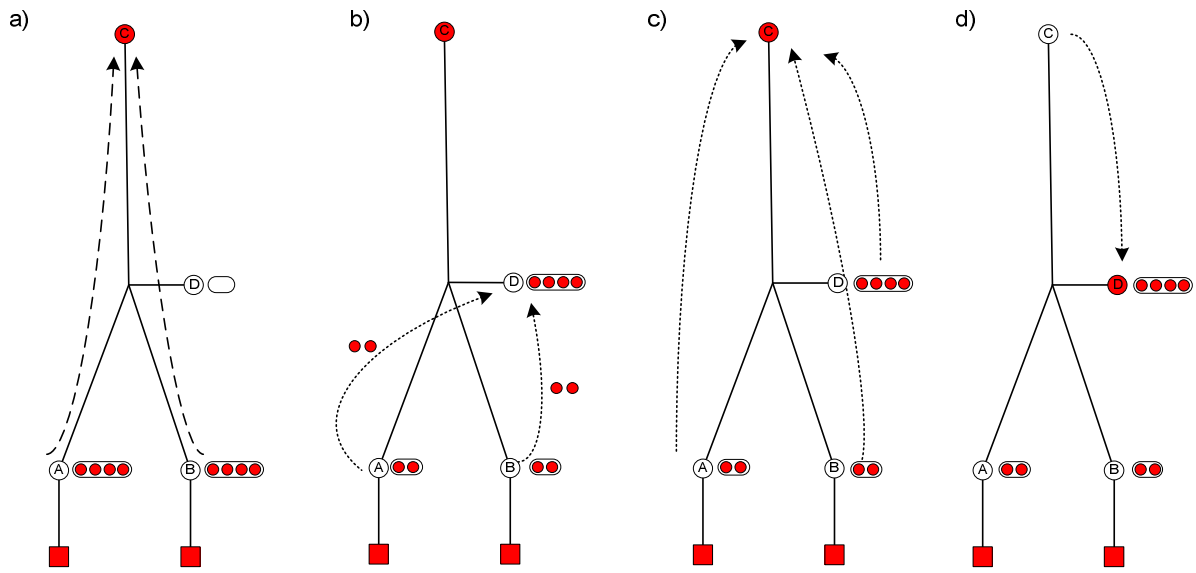


Fig 6.6. Coins exchange

6.5.2.3. Coin list analysis and replica management

In this activity the coins list is counted up. The list of files on which replicas are stored locally is supplemented with every remote file, which has at least one coin on the coin list. The demand for each file on the list is calculated according to the *evaluation function*:

$$eval_f = \sum_{c \in C_f} \left(read(c) + diss(c) \cdot NVN(c) write(c) \right) \qquad (6.3)$$

Where:

- $eval_f$ – the evaluation function for file *f*,
- C_f – the set of coins related to file *f*,
- $read(c)$ – the function which returns *1* if coin *c* was generated as a result of read operation and *0* otherwise,
- $write(c)$ – the function which returns *1* if coin *c* was generated as a result of write operation and *0* otherwise,
- $NVN(c)$ – the number of storage servers that the coin visited,
- $diss(c)$ – the *dispersion factor* describes how many different paths the coin could traverse. Initially, the factor is 1, then in every storage server it visits the factor is multiplied, with the number of neighbors of the visited node.

The strategy improves the influence of the coins from other nodes. The list of files is then sorted by the value of the evaluation function, so the most often used files are at the beginning. Then some of the first and last elements from this list are taken into further consideration. The number of the first elements are given by the *replication factor* (*RF*) and it is expressed as a percentage of total list size. Similarly, the number of last elements is given as the *dereplication factor* (*DRF*). For every remote file from the first elements the replication request is generated and sent to the client, from which most of the coins related to the file was from. The replication request contains the value of the evaluation function calculated locally

and the id of the requestor. On the list of files to the dereplication there are the local files selected from the last elements.

The next step is the processing of the replication requests, which came to the host since the last launch of the algorithm. The *replica candidate* is a storage server, which sent a request. The *acceptance factor* (*AF*) is a value, which describes the threshold of replica candidate acceptance. The requestor is granted with the replica if its request evaluation function is not lower than the evaluation function for the requested node, calculated in the current run of the algorithm, multiplied by *AF*.

The replica is moved to the replica candidate if and only if it was granted with a replica and any of the following occurred:

- the local replica is on the list of files to the dereplication,
- if the value of the evaluation function computed by the replica candidate multiplied by the *move factor* (*MF*) is bigger than the corresponding value calculated in the requested node.

If the replica should be moved to more than one node, it is moved to the first one, and the new replica is installed in the others. To avoid a situation in which the replica is moved to the other server and replicated back to the original server, the AF and MF parameters must preserve condition:

$$AF \cdot MF \geq 1 \tag{6.4}$$

To prevent overlapping of the lists of desired replicas and of files to dereplication the RF and DFS parameters must fulfill the following condition:

$$RF + DRF \leq 1 \tag{6.5}$$

The files that are on the list of files to the dereplication, which were not moved, are removed from the local storage, unless the number of existing replicas is less than NOR_{min}. The created replica is pinned for the time equals TTL_c defined for each file. During this time the replica cannot be moved or dereplicated.

6.6. Summary

The SoRPA can be adjusted with several parameters and strategies. We can divide them into two groups:

- provided to the system administrator as a configuration parameters: TTL_c, NOR_{min}, p_c
- the algorithm parameters for which optimal values must be found: *k, RF, DRF, AF, MF*

The values and the influence of the parameter from the first group can be easily predicted. The parameters that form the second group must be investigated to choose the best values for them.

The workload balancing, the disk drive's performance and the utilization of free space among the storage servers are not considered by the proposed algorithm and may influence overall system performance. As far as a disk size may be a serious case in the real-life applications for the performance tests, we can provide enough free space on the disk in each node to store all possible replicas. The rules of the replica request generation and request acceptance can be extended to consider a utilization of free space among the storage servers. The algorithm's aim to place the replica in the node, which is the best localization for the copy. It does not analyze if the node has enough performance to serve the request to all replicas that it holds. Similar to the disk size, we are able to provide for the test enough resources for nodes so they can process all requests. The rules of the replica request generation and request acceptance rules can also be extended to include node performance bounds, but these issues are out of the a scope of this work.

There are two services, which are utilized by the algorithm and that can strongly influence the system performance: the network discovery service and the replica discovery service. The network discovery service is responsible for inspecting networks links to the other nodes and selecting the closest neighbors. Measuring connection efficiency by all nodes to all other nodes is time and resource consuming. For the algorithm performance the most important is to sort nodes in link quality order, rather than knowing an exact link latency for each node. The nodes with the fastest links should be measured more accurately then more distant nodes. The SoDFS use the network discovery module, which measures the link efficiency of all other nodes. It is enough for a small number of nodes, which are used for evaluation in this work. However the network pooling service is a part of the system that may be investigated by further research, to make it applicable for a larger number of nodes.

The replica discovery service is responsible for providing information about replica localization. When a new replica is created the nodes that are interested in using this copy should switch to use this copy as fast as possible. The meta-server is the SoDFS replica discovery service. The SoDFS analyzes the replicas list periodically and chooses the closest as a copy on which read or write will be performed. This information should be cached locally by each node, otherwise the meta-server would be flooded by requests. The cache refresh policy can cause serious side effects for algorithm performance.

The SoRPA is a design for the requirements of SoDFS, however it can be adapted to another system that brings replicas of their resources closer to the clients.

7. Evaluation

The goal of the evaluation is to find the optimal values for the SoRPA parameters and check whether the algorithm exhibits the desired properties (Section 6.36.3) in the found setting. The algorithm has eight parameters of TTL_c, NOR_{min}, p_c, k, RF, DRF, AF, MF. The first three are the administrative parameters, which values can be easily determined. The coin time to live (TTL_c) characterizes the period, from which the workload is taken to the analysis. In the following measurements its value is set to 60 seconds for every file. In the survey every file has at least one replica ($NOR_{min} = 1$). The probability of coin creation (p_c) should be proportional to the predicted workload. The number of coins must be sufficient for the analysis and should not use too much memory. In the study $p_c = 0.8$.

The five remaining parameters (k, RF, DRF, AF, MF) are investigated in three experiments. Since the goal of the two first experiments is to examine the replication mechanism based on the replication request and orders, they do not involve the coin exchanges ($k = 0$). The first one evaluates the correlation of RF to DRF. The AF and MF are predetermined. The second experiment investigates the relation between AF and MF. The values of RF and DRF are set to the best values found in the first experiment. The third experiment evaluates the impact of the coin exchanges mechanism on the ability of an algorithm to deal with a read/write optimization problem (Section 6.4.1.). The results of the last experiment may be influenced by the algorithm determined by the closest neighbor. (Section 5.2.3.5.). The investigation on the algorithm impact was left for further research.

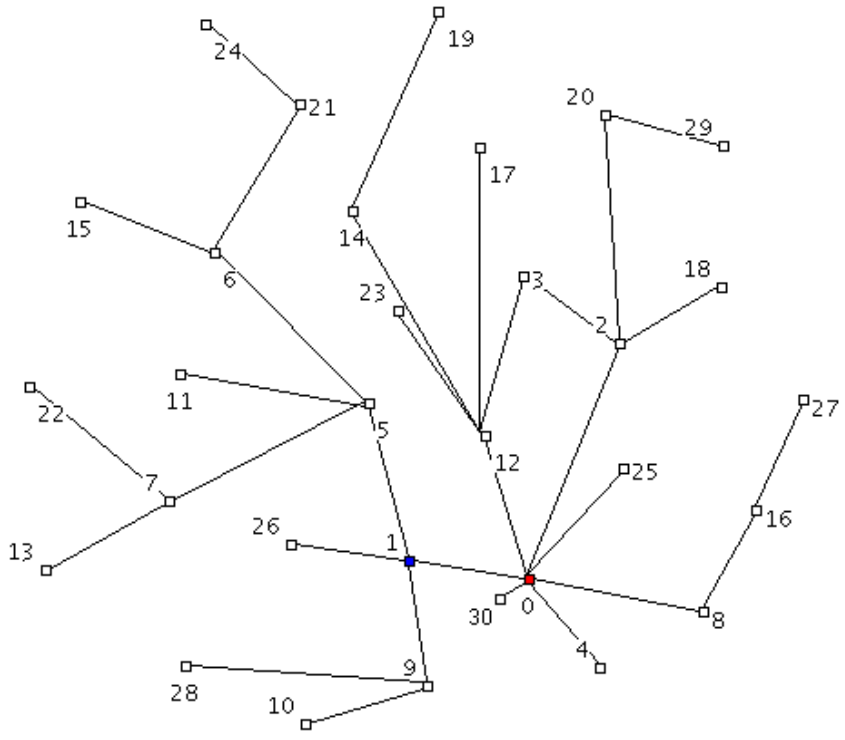


Fig 7.1. Test network topology

The goal of the SoRPA algorithm is to reduce the client perceived latency in the file access. The metric used to evaluate algorithm efficiency is the number of operations per second performed by all clients of the system. The metric gives results proportional to the delays perceived by the client. When a client executes operations in a sequence, the lower latencies allow clients to perform more operations in the same period.

The experiments were performed in the network emulation testbed created at the University of Stuttgart. The environment utilizes the so-called time jails approach to the network emulation. It was developed at the University of Stuttgart and introduced by Grau et al. in [14].

The simulation environment consists of five machines, which simulate 31 virtual nodes. The nodes represent the servers in the 31 autonomous systems connected with the network generated with a BRITE topology generation tool [25], [43], using Waxman's algorithm. The link latencies are set to 5 milliseconds. The network topology is presented in the Fig 7.1.

Due to the memory usage problem not all nodes could contain the storage server, as it was primarily considered. Only eleven nodes: 2, 3, 6, 11, 13, 19, 24, 26, 27, 28, 29 hold the storage servers. Nodes 18, 12, 15, 5, 7, 14, 21, 9, 16, 10, 20 run the *executors*. Each of them starts three clients, which generate test traffic in the corresponding storage servers. Node 30 contains the meta server and node 1 runs its database. On every node three files are created. It gives 33 files overall. The files are initially written with 30 KB of random data. The file size is small to reduce the impact on the file state transfer during replication. Each client picks one of the 33 files randomly, opens it and performs from 1 up to 300 operations. The number of the operation is arbitrary. Each operation reads or writes 4KB from or to a random place in the file. According to the evaluation of the Andrew File System (the ancestor of Coda) presented in [29], there are 24% of the writes and 76% of the reads in the tested environment. To make the measurements from the two first experiments compatible with the third one we assume that writes are 25% of the operations in them.

7.1. Experiments

7.1.1. Reference measurements

The reference measurements were taken with the turn off mechanism of replication. They measure the behavior of the system for a different factor of write operation (*WF)*. The *WF* varies from 0 to 1 with the step of 0.25. As we can see in the Fig 7.2. the results are almost the same for different values of the factor. This is expected behavior, since there is only one copy of each file and no update propagation is needed. As a consequence the cost of a write is near to the cost of a read. As we can see, the time needed for writing small pieces of data to the hard drive (which is significantly larger than the time required for reading) does not count as much as the network delays.

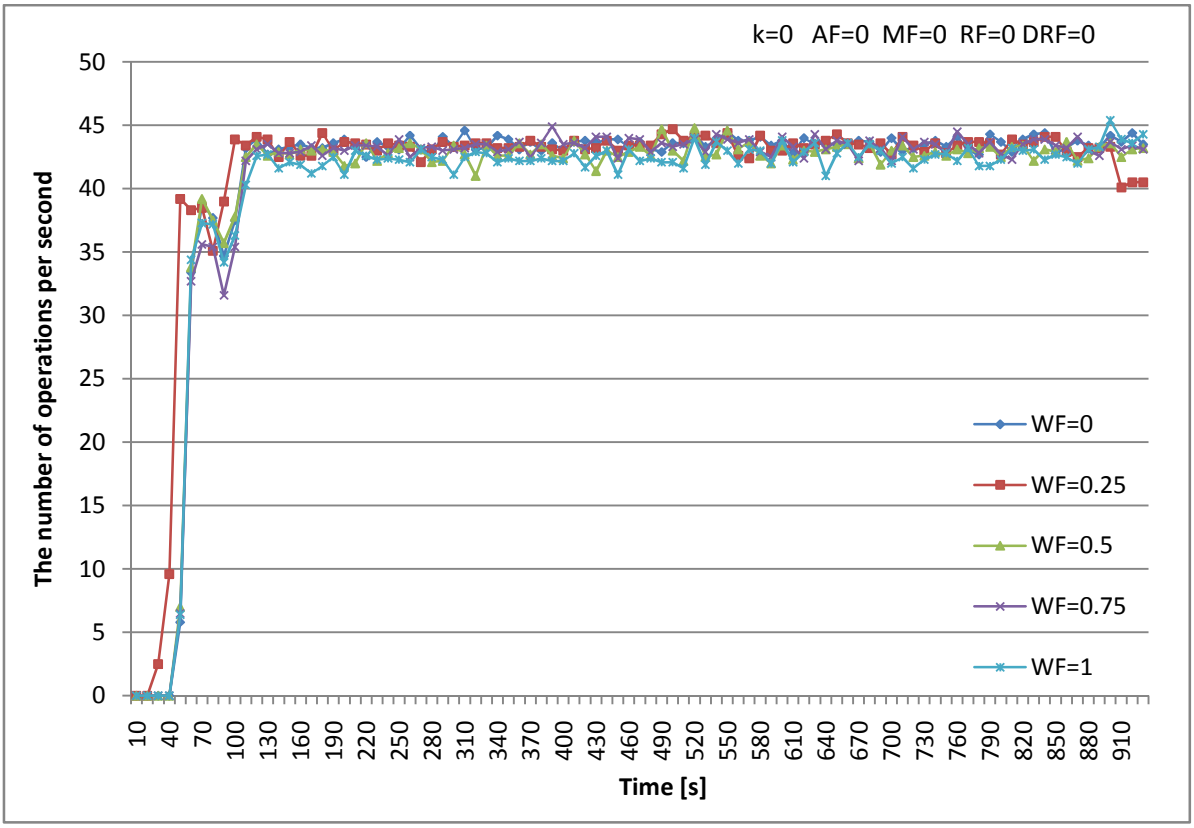


Fig 7.2. Reference measurements

7.1.2. Experiment 1 – *RF* and *DRF*

The experiment investigates the correlation between *RF* and *DRF* parameters. The measurements were taken for the following pairs of the parameters *(RF; DRF) = (0.6; 0.1), (0.6; 0.2), (0.6; 0.3), (0.6; 0.4), (0.7; 0.1), (0.7; 0.2), (0.7; 0.3), (0.8; 0.1), (0.8; 0.2), (0.9; 0.1).* The values of AF and MF were set to the fix values, respectively: 0.7 and 1.45. The results are presented in the Fig 7.3. The measurements initially fluctuate strongly, then they stabilize on different levels, depending on the *RF* and *DRF* parameters. We suspect two reasons for strong initial oscillation. Since it is also observed in the reference measurement, the possible explanation is a not the simultaneous start of the traffic generators. The second reason gives the explanation for a much stronger variation than the one observed in the reference measurements. It is probably the replication mechanism, which changes a lot in the replica placement at the beginning that is to blame.

The rule observed is, the bigger the *RF* the higher stable results that are achieved. This is expected behavior. However there are exceptions from that rule. The *RF*=0.7 and *DRF*=0.2 gave the worst results. Even *RF*=0.6 *DRF*=0.1 is better. There is no rule for the impact of *DRF*. In the case that *RF*=0.8 the *DRF*=0.1 gave better results than the *DRF*=0.2. In contrast, for the *RF*=0.6 the best results are achieved for the highest value of *DRF* possible i.e. 0.4.

The best results were achieved for the *RF*=0.9 and *DRF*=0.1. The stable results are on average 15% higher than the reference measurement. The comparison is presented in the Fig 7.4.

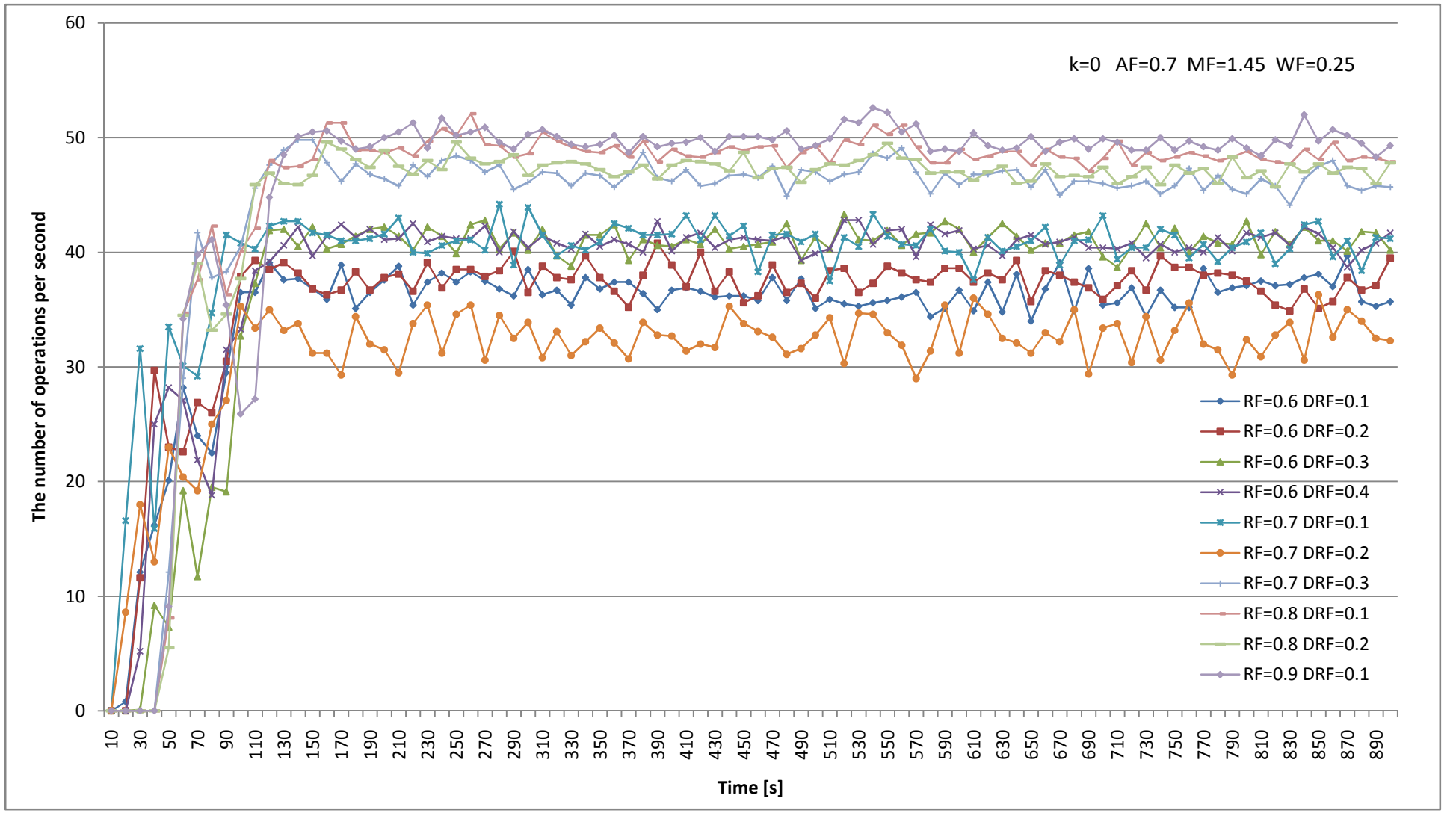


Fig 7.3. *RF* and *DRF* correlation

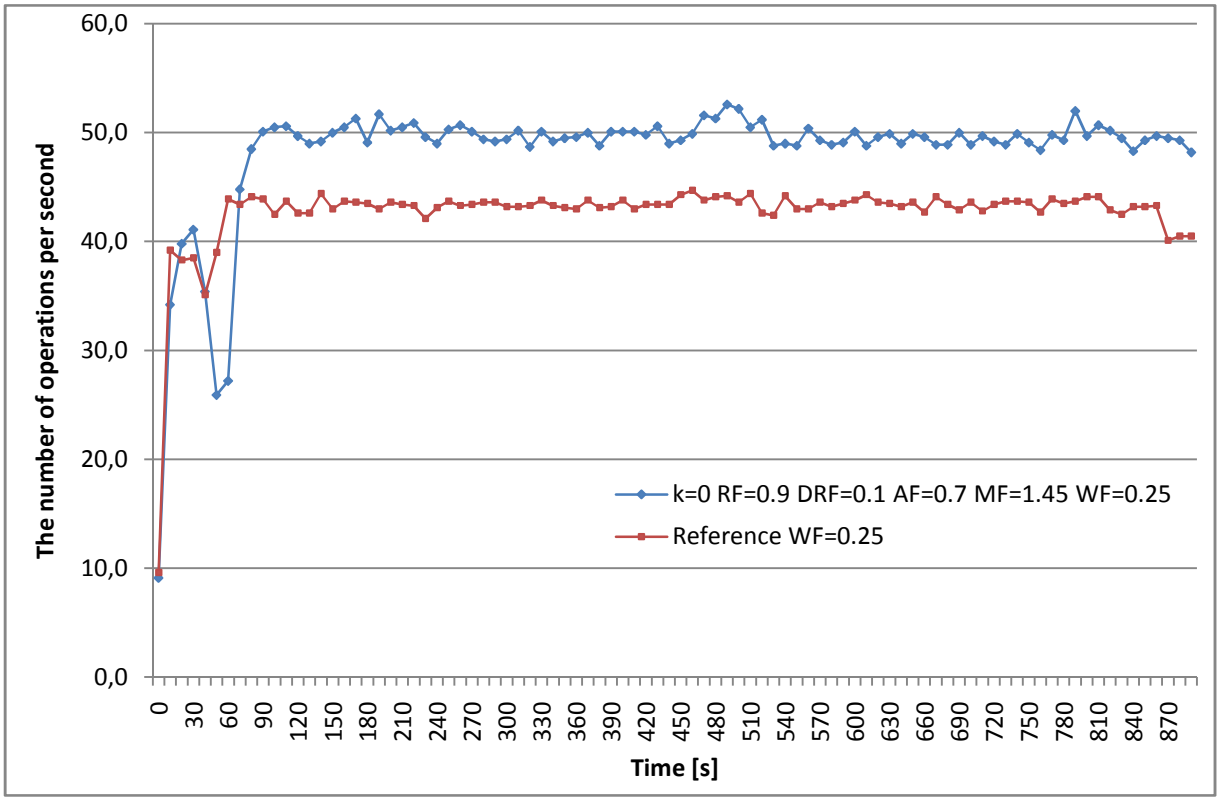


Fig 7.4. Results for *RF*=0.9 and *DRF*=0.1 compared to the reference measurement

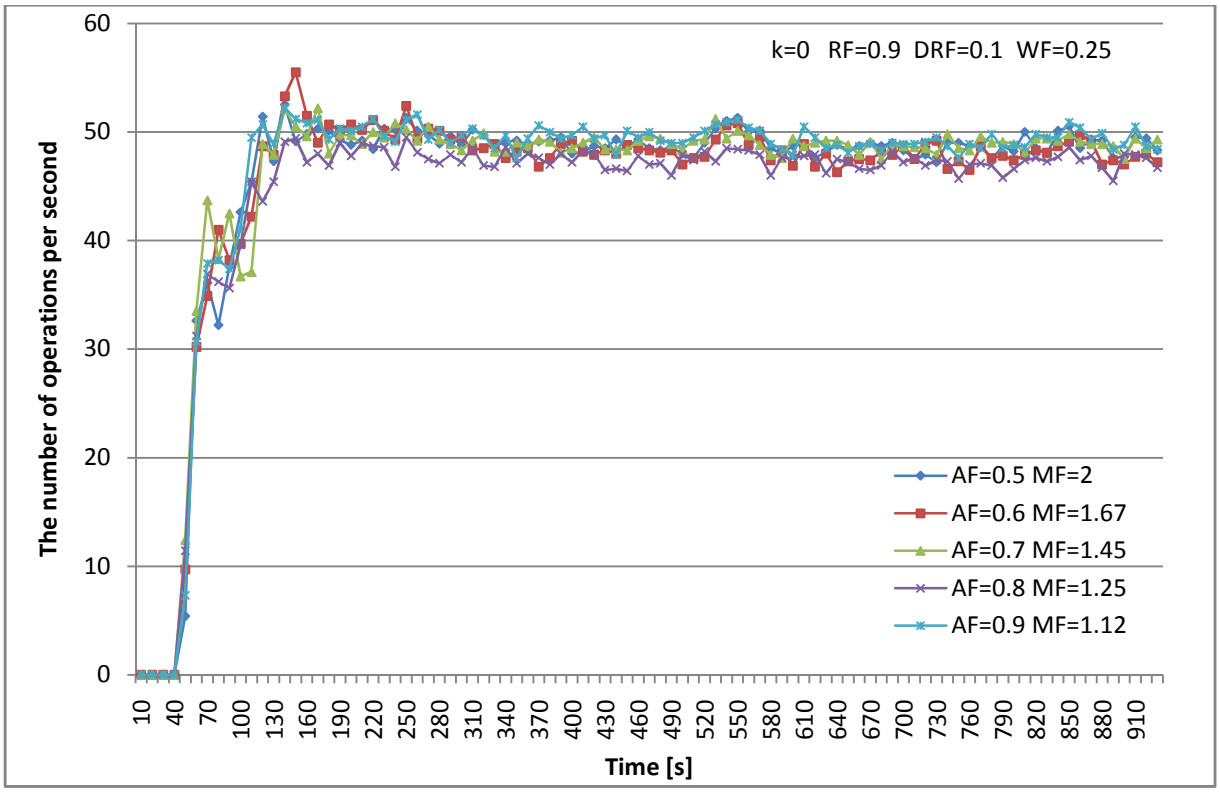


Fig 7.5. *AF* and *MF* correlation

7.1.3. Experiment 2 – *AF* and *MF*

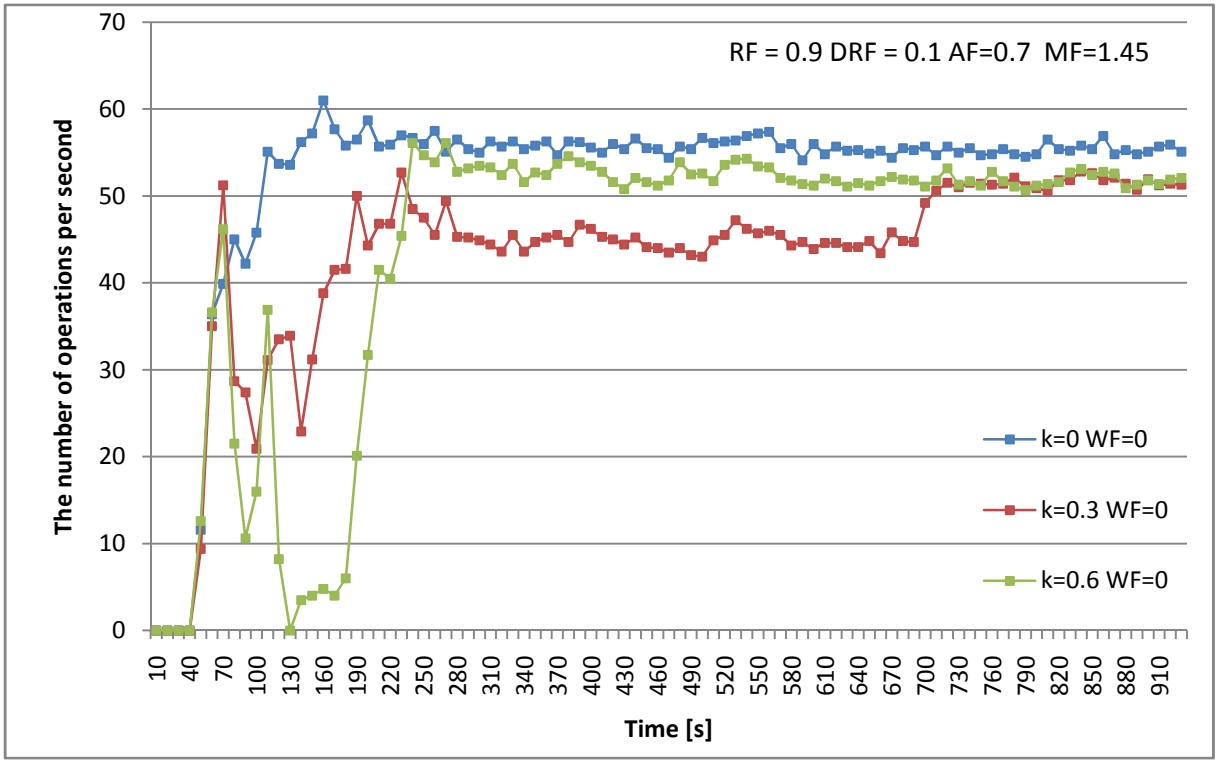
The second experiment examines the correlation between AF and MF parameters. The values of RF and DRF were set to the best values found in the first experiment (RF=0.9 and DRF=0.1). The measurements were conducted with the following pairs of the parameters *(AF; MF) = (0.5; 2), (0.6; 1.67), (0.7; 1.45), (0.9; 1.12)*. The applied value of MF is the smallest one, which fulfills the (6.4) condition. The greater values do not bring better results, so they were not taken into consideration.

The results, which are presented in the Fig 7.5. are different than expected. They stabilize at a similar level, regardless of the parameters applied. It seems that the test environment is too small to measure the impact of those parameters. Probably, a greater number of storage servers and files could reveal the differences.

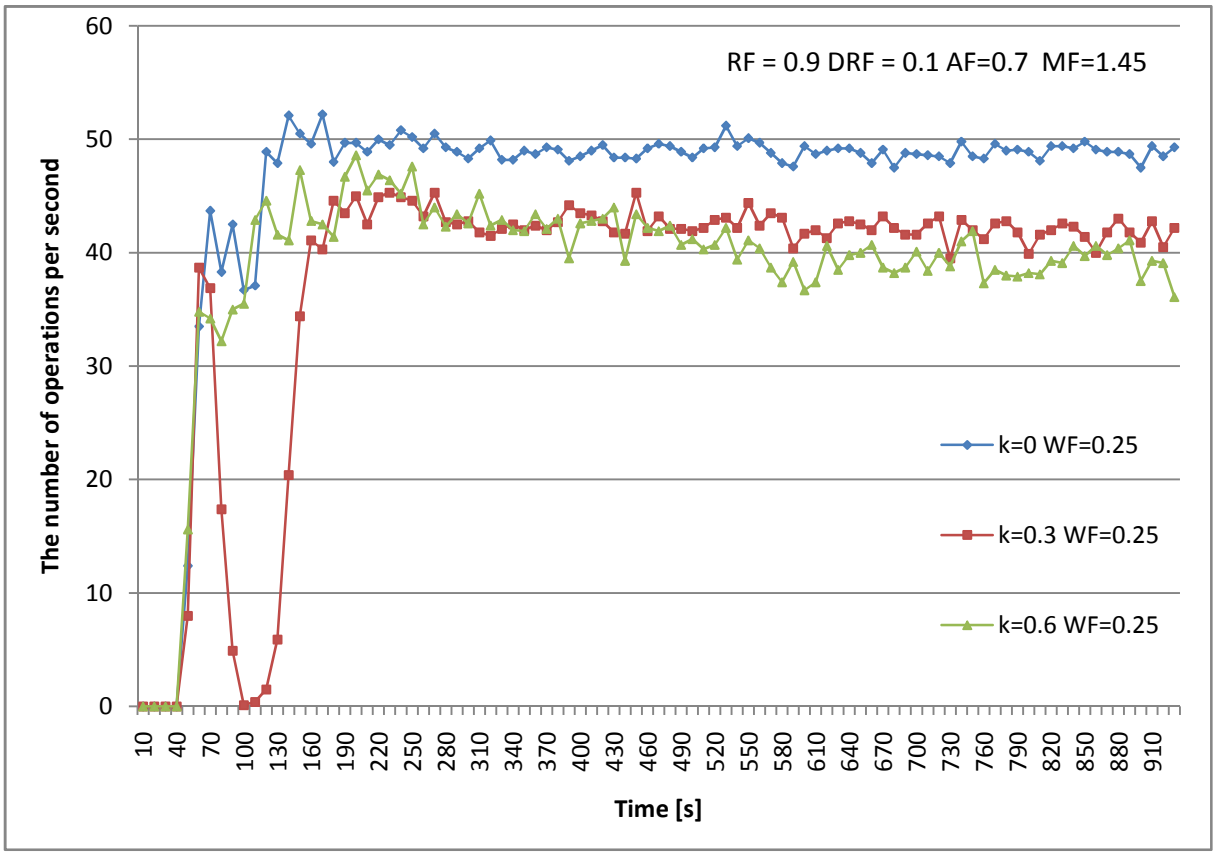
7.1.4. Experiment 3 – *k* and writes percent

The third experiment examines the coins exchange mechanism. The parameter *k* is set to 0, 0.3 or 0.6 and the algorithm is tested under the workload, which varies WF from 0 to 1 with the step of 0.25. The values of RF and DRF were set to the best values found in the first experiment. Since the values of the AF and MF parameters do not have a significant impact on performance, the values *(AF; MF) = (0.7; 1.45)* where chosen arbitrarily.

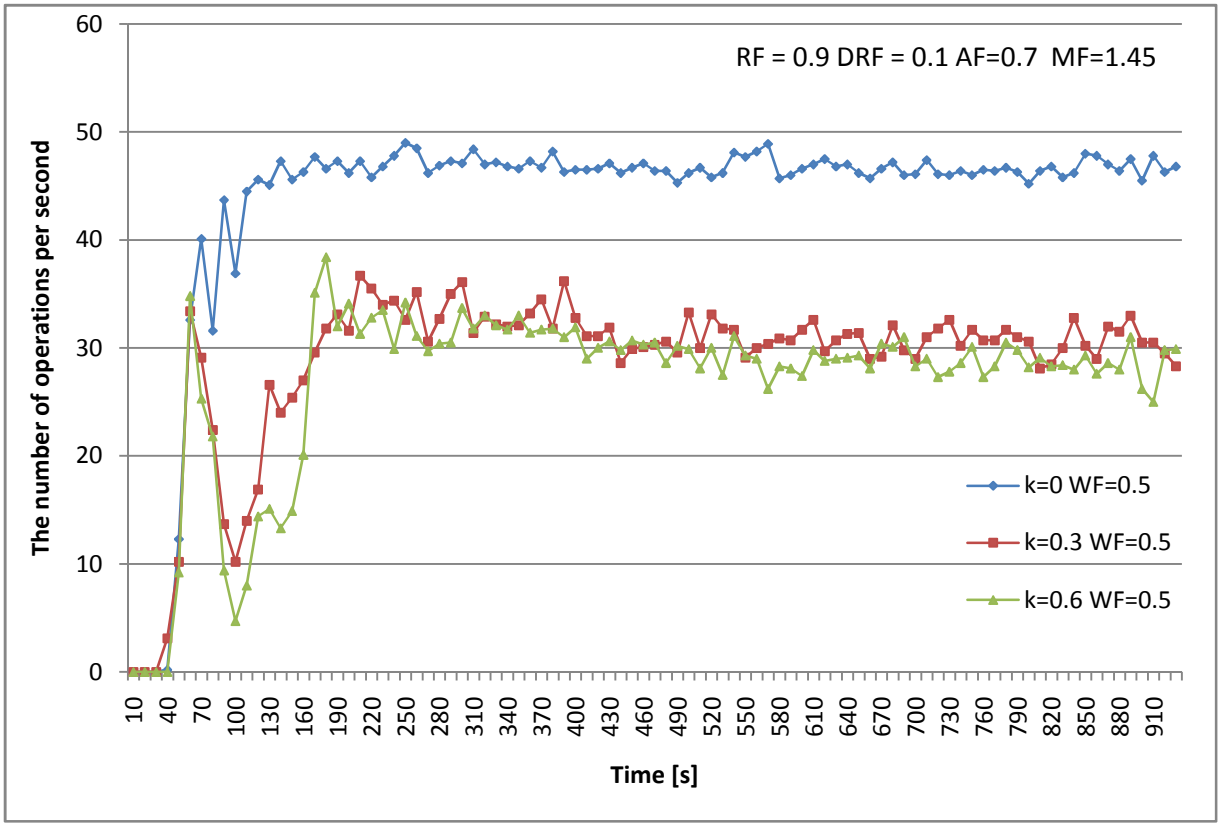
As we can see in the Fig 7.6.a-e), the coins exchange mechanism does not improve reads/writes optimization. In fact it has only a negative impact, reducing performance significantly. The coins exchange mechanism was intended to gather the coins between the nodes that utilize a file. It seems, that the mechanism disperses the coins in the entire network instead. In consequence it leads to the slowdown.



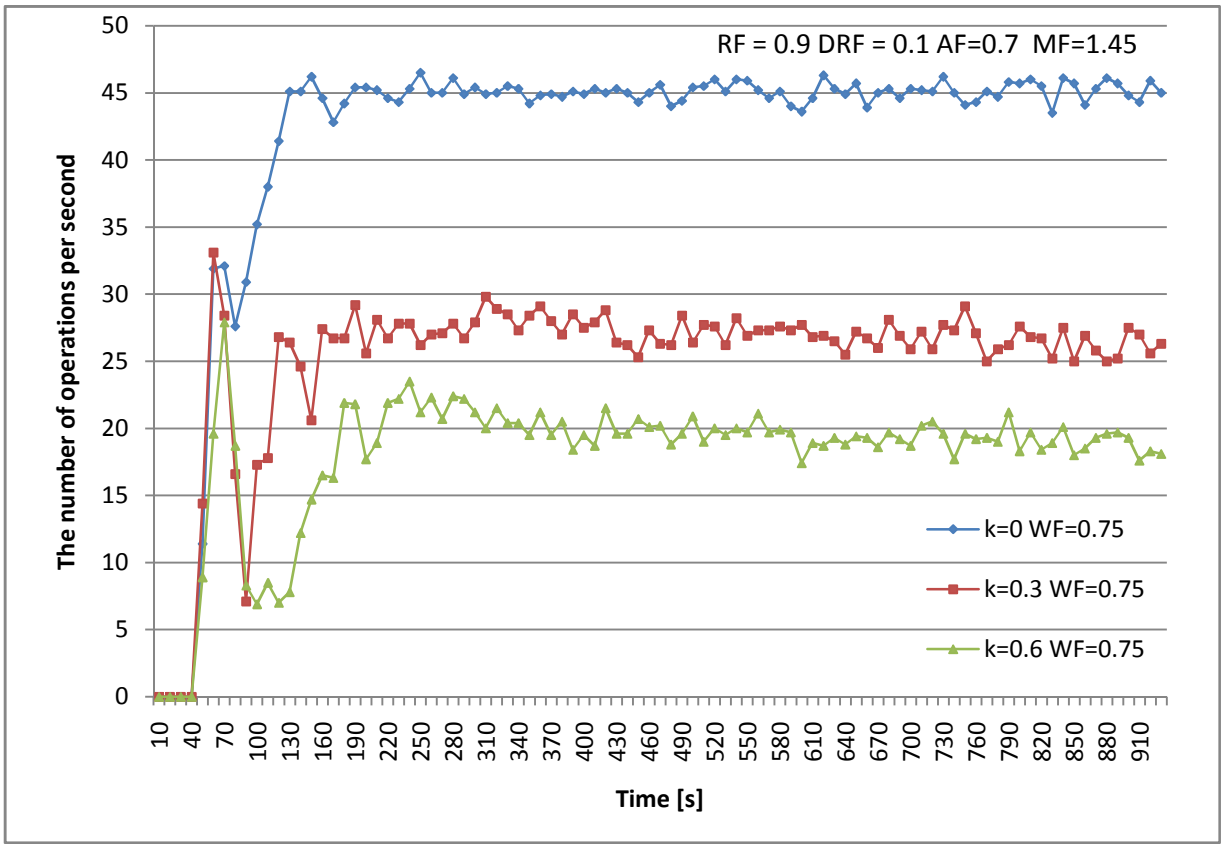
a) *WF*=0



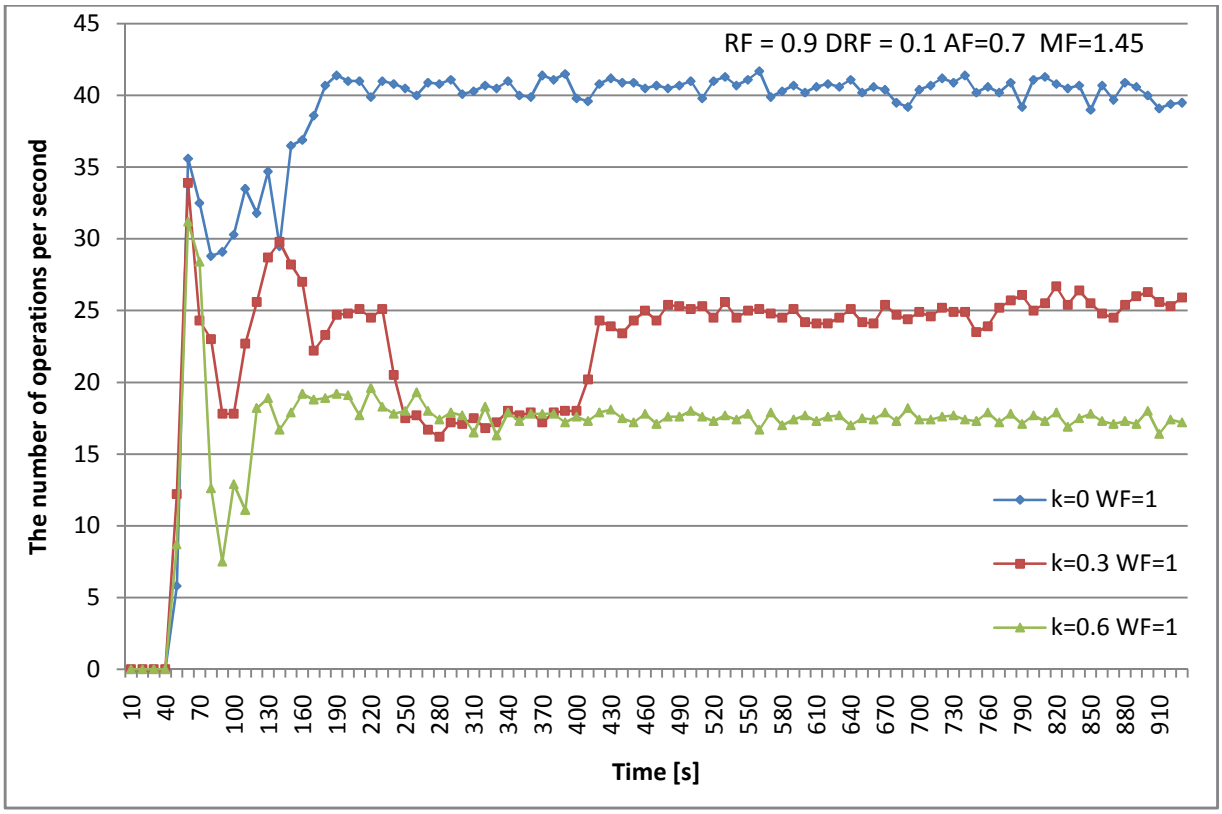
b) *WF*=0.25



c) *WF*=0.5



d) *WF*=0.75



e) *WF*=1

Fig 7.6. Influence of *k* parameter

8. Summary

The goal of the work was to create a distributed file system. It was successfully accomplished and the Self-organizing Distributed File System was created as a flexible platform for testing various replica placement algorithms.

The proposed Self-organizing Replica Placement algorithms, explore the prospect of utilizing a self-organization phenomenon to accomplish the second main goal of the work. Although the reached speed up is not very high, the algorithms exhibit the desired stabilization property. This means that the replication mechanism is based on the exchanges of the replication request and orders works.

Unfortunately, the evaluation showed that the algorithm does not show all the desired properties. The proposed mechanism of coins exchanges failed in providing the reads/writes optimization. Also we were unable to check dynamic properties of the algorithm. The impact of *AF* and *MF* must be investigated in more detailed tests, which would involve a more localized workload and greater sets of storage servers and files. The problems encountered during the experiments suggest utilizing simulation instead of a test over a real system.

The gathered results encourage further research. The main study should be focused on finding the substitution for the coin exchange mechanism. The intended behavior of the mechanism was to place coins in the storage servers between the nodes, which utilize the file. Then by assigning the higher weight to the exchanged writes coins, the replicas would be fetched to these servers if the number of the writes operation would become significant, or the replicas would be spread closer to the clients in the case of an increasing number of reads. The new mechanism of placing the writes coins in the intermediate storage server could put them on the paths of update propagation. However, the multicast protocols working in the application layer have the same problem as the replica placement algorithm. Knowledge about the network structure is crucial for their performance, but gathering it might be expensive. The issue of the creation of a self-organizing replica placement algorithm is equal to the problem of designing the self-organizing multicast protocol.

9. Index of Figures

10. Index of Tables

11.References

[1] Anderson T., Dahlin M., Neefe J., Patterson D., Roselli D., Wang R., *Serverless network file systems.* ACM SIGOPS Operating Systems Review, Volume 29 , Issue 5 (December 1995) pp. 109 – 126

[2] Arlitt M., Jin J., *Workload characterization of the 1998 world cup web site.* IEEE Network, Vol. 14, No. 3, pp. 30-37, May/June 2000

[3] Birman K., Joseph T., *Exploiting Virtual Synchrony in Distributed Systems*. Proceeding of the 11th Symposium on operating system principles, pp. 123-138, ACM 1987

[4] Birman K., Joseph T., *Reliable Communication in the Presence of Failures.* ACM Transactions on Computer Systems, February 1987, 5, 1, pp. 47-76

[5] Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Chandra T., Fikes A., Gruber R., E., *Bigtable: A Distributed Storage System for Structured Data.* ACM Trans. Comput. Syst., Vol. 26, No. 2. (June 2008), pp. 1-26

[6] Deamers A., Greene D., Hauser C., et al, *Epidemic algorithms for replicated database maintenance.* Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, 1987

[7] Dean J., Ghemawat S., *MapReduce: Simplified Data Processing on Large Clusters.* OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004

[8] Dorigo M., *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992

[9] Dowdy L. W., Foster D. V., *Comparative Models of the File Assignment Problem.* ACM Computing. Surveys, Vol. 14, No. 2. (1982), pp. 287-313

[10] Dubois M., Scheurich C., Briggs F., *Synchronization, Coherence, and Event Ordering in Multiprocessors. IEE Computer,* February 1988, 21, 2, pp. 9-21

[11] Eswaran K. P., *Placement of records in a file and file allocation in a computer network.* IFIPS Conference Proceedings, pp. 304-307, 1974

[12] Floyd S., Jacobson V., Liu C.-G., et al, *A reliable multicast framework for light-weight sessions and application level framing*. IEEE/ACM Transactions on Networking, 5(6), December 1997 5

[13] Ghemawat S., Gobioff H., Leung S-T., *The Google File System.* In The Proceedings of the Symposium on Operating Systems Principles, Bolton Landing, NY, USA, October 2003

[14] Grau A., Maier S., Herrmann K., Rothermel K., *Time Jails: A Hybrid Approach to Scalable Network Emulation*. 22nd Workshop on Principles of Advanced and Distributed Simulation, 2008, pp. 7-14

[15] Guy R. G., Heidemann J. S., Mak W., Thomas W. Page Jr., Popek G. J., Rothmeier D., *Implementation of the Ficus Replicated File System.* In Usenix Conference Proceedings, June 1990

[16] Hutto P., Ahamad M., *Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories.* Proceeding of the Tenth International Conference on Distributed Computing Systems, pp. 302-311, IEEE 1990

[17] Kangasharju J., Roberts J.,. Ross K. W., *Object replication strategies in Content Delivery Networks.* Proceedings of the 5th ACM SIGMM international workshop on Multimedia information retrieval Berkeley, California pp. 255 – 261, 2003

[18] Karlsson M., Karamanolis Ch., Mahalingam M., *Framework for Evaluating Replica Placement Algorithms.* Technical Report HPL-2002, HP Laboratories

[19] Karlsson M., Mahalingam M., *Do We Need Replica Placement Algorithms in Content Delivery Networks?* published in and presented at Web Content Caching and Distribution (WCW) 2002, 14-16 August 2002, Boulder, Colorado

[20] Lamport L., *Concurrent Reading and Writing of Clocks*. ACM Transactions on Computer Systems, November 1990, 8, 4, pp. 305-310

[21] Lamport L., *How to Make a Multiprocessor Computer that Correctly Executes Multiprocessors Programs*. IEEE Transactions on Computers, September 1979, 29, 9 pp. 690-691

[22] Lamprot L., *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, July 1978, 21, 7, pp. 558-565

[23] Lipton R., Sandberg J., PRAM: *A Scalable Shared Memory*. Technical Report CS-TR-180-88. Princeton University, September 1988

[24] Mahmoud S. and Riordon J., *Optimal Allocation of Resources in Distributed Information Networks.* ACM Transactions on Database Systems, vol. 1, no. 1, pp. 66-68, March 1976

[25] Medina A., Lakhina A., Matta I., Byers J., *BRITE*: *An Approach to Universal Topology Generation*. In Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, August 2001

[26] Obraczka K., Silva F., *Network Latency Metrics for Server Proximity*. Proceedings of the IEEE Global Telecommunications Conference, 2000, GLOBECOM, pp.421 - 427

[27] Popek G. J., Guy R. G., Page T. W. Jr., Heidemann J. S., *Replication in Ficus Distributed File Systems.* Proceedings of the Workshop on Management of Replicated Data, pp. 20-25. 1990

[28] Rabinovich M., Rabinovich I., Rajaraman R., Aggarwal A. *A Dynamic object Replication and Migration Protocol for an Internet Hosting Service.* Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, p. 101, 1999

[29] Spasojevic, M., Satyanarayanan, M., *An Empirical Study of a Wide-Area Distributed File System*. Technical Report. Transarc Corporation. Nov. 1994

[30] Tanenbaum A. S., van Steen M., *Distributed Systems Principles and Paradigms.* Prentice Hall, 2007

[31] Tatebe O., Morita Y., Matsuoka S., Soda N., Sekiguchi S., *Grid Datafarm Architecture for Petascale Data Intensive Computing*. CCGRID, Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002, p.102

[32] Tatebe O., Sekiguchi S., Morita Y., Matsuoka S., Soda N., Worldwide Fast File Replication on Grid Datafarm. Proceedings of the 2003 Computing in High Energy and Nuclear Physics, La Jolla, Ca, USA, March 2003

[33] Tatebe O., Sekiguchi S., Morita Y., Soda N., Matsuoka S., Gfarm v2: A Grid file system that supports high-performance distributed and parallel data computing. Proceedings of the 2004 Computing in High Energy and Nuclear Physics , Interlaken, Switzerland, September 2004

[34] Terry D., Demers A., Petersen K., Spreitzer M., Theimer M., Welsh B., *Session Guarantees for Weakly Consistent Replicated Data.* Proceeding of the Third International Conference on Parallel and Distributed Information Systems, pp. 140-149, IEEE 1994

[35] Terry D., Petersen K., Spreitzer M., Theimer M., *The Case for Non-transparent Replication: Examples from Bayou*. IEEE Data Engineering, December 1998, 21, 4, pp. 12-20

[36] Wolfson O., Jajodia S., Huang Y., *An Adaptive Data Replication Algorithm.* ACM Transactions on Database Systems, vol. 22, pp. 255-314, 1997

[37] http://db.apache.org/derby/http://db.apache.org/derby/

[38] http://hadoop.apache.org/core/

[39] http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[40] http://technet2.microsoft.com/windowsserver/en/library/d3afe6ee-3083-4950-a093-8ab748651b761033.mspx?pf=true

[41] http://www.wuala.com/

[42] http://www.alfresco.com/products/aifs/

[43] http://www.cs.bu.edu/brite/

[44] http://www.jgroups.org/

[45] http://www.oracle.com/technology/products/ias/toplink/index.html

[46] http://www-unix.mcs.anl.gov/romio/

[47] http://www.youtube.com/watch?v=3xKZ4KGkQY8 (Google Tech Talks)

Appendix A – Contents of the attached DVD

- The attached DVD disc contains the following directory tree:
- `sodfs\` – the entire environment installed on every node of the testbed,
- `sodfs\jlan\` – the storage server, the meta-data server,
- `sodfs\jgroups\` – the gossip router required by JGroups,
- `sodfs\derby\` – the Apache Derby database utilized by the meta-data server,
- `sodfs\testclient\` – the test client of the system and tools coordinating its work,
- `sodfs\uploads\` – the directory utilized for uploading the files to the testbed,
- `sodfs\downloads\` – the directory utilized for downloading the files to the testbed,
- `sources\` – the source code for SoDFS and the test client,
- `experiments\` – the results of the experiments and a reference measurements,
- `document\` – this document in various formats.

Załącznik nr 1 do ZW 23/2001

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ INFORMATYKI
I ZARZĄDZANIA
Instytut Informatyki I-32

Roman Kierzkowski

Temat pracy dyplomowej:

SAMOORGANIZUJĄCY SIĘ ROZPROSZONY SYSTEM PLIKÓW.

O Ś W I A D C Z E N I E

Wyrażam zgodę na udostępnienie mojej pracy dyplomowej

.....
(podpis)

Wrocław, dnia 26. stycznia 2009 r.

Roman Kierzkowski
numer albumu
133666

Oświadczenie

Stwierdzam, że przedłożoną na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej pracę magisterską, na kierunku Informatyka pod tytułem:

SAMOORGANIZUJĄCY SIĘ ROZPROSZONY SYSTEM PLIKÓW

opracowała(e)m samodzielnie. Wszystkie przytoczone w pracy teksty dosłowne innych autorów udokumentowane zostały w formie cytatów, natomiast dane, stwierdzenia i poglądy autorów przytoczone niedosłownie opatrzone zostały odpowiednimi odsyłaczami.

Praca ta nie była wcześniej publikowana i przekładana do jakiejkolwiek oceny.

Wrocław, dnia 26. stycznia 2009 r.

.....

podpis studenta