UNIVERISTY OF CYPRUS
DEPARTMENT OF COMPUTER SCIENCE

IMPLEMENTATION AND EXPERIMENTAL EVALUATION OF THE LAYERED DATA REPLICATION ALGORITHM

MASTER OF SCIENCE

MARILENA DEMETI

2013

IMPLEMENTATION AND EXPERIMENTAL EVALUATION OF THE LAYERED DATA REPLICATION ALGORITHM

Marilena Demeti

A Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
at the
University of Cyprus

Recommended for Acceptance
by the Department of Computer Science
January, 2013

ABSTRACT

In this thesis, we provide an implementation and experimental evaluation of the LDR algorithm [13]. LDR has the potential of being the basis of a distributed file system that achieves the three main goals that any File Distributed System wish to achieve, which are replication, performance and consistency guaranties and all these without using any high level functions such as distributed locking [18], embed physical writes to the data within a logical read [1, 19] or group communication [21], that negatively affect performance [22].

Our implementation can be used both in LAN and WAN, giving writers the opportunity to write all common types of files, including .doc, .pdf, .txt as well as .jpg images, and readers the certificate that they read the most up-to-date version of a file.

The experimental evaluation took place on the Planetlab platform, and demonstrated that for up-to 250KB file size and a concurrent use from 10 writers and 20 readers, the system manages to serve all the clients and keeps its performance in high levels, using messages of 944 bytes as a default block size.

Our expectations were fulfilled. As the results showed the implementation works for unbounded number of clients without violating the consistency guaranties and by having satisfying execution performance for large data objects replication.

Approval Page

Master Dissertation

IMPLEMENTATION AND EXPERIMENTAL EVALUATION OF THE LAYERED DATA REPLICATION ALGORITHM

Presented by

Marilena Demeti

Research Consultant	Dr. Chryssis Georgiou
Committee Member	Dr. Nikolas Nikolaou
Committee Member	Dr. Vasos Vasiliou

University of Cyprus

January, 2013

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my supervisor professor, Dr. Chryssis Georgiou, for the guidance and helpful advices, he gave me during the preparation of this thesis. I also would like to thank Dr. Nikolas Nikolaou for his helpful advises and assistance every time I called for his help.

I would also like to thank my parents Andreas Demetis and Christa Rousou Demeti as well as my good friends Maria Philippou and Eleni Philippou for the useful advices, patience and love they showed me all this time.

Dedicated to my father Andreas Demetis.

TABLE OF CONTENTS

Chapter 1: Introduction **1**

1.1 Motivation and Prior Work 1

1.2 Contribution 3

1.3 Thesis Organization 4

Chapter 2: Background **5**

2.1 Atomic R/W Registers 5

2.2 Other Distributed File Systems 8

2.2.1 Network File System 8

2.2.2 Coda File System 10

2.2.3 Plan 9 14

2.2.4 xFS: File System without Server 17

2.2.5 Federated Array of Bricks 18

2.2.6 Google File System 19

2.2.7 Belisarius 22

2.3 Sockets 23

2.4 Transport Layer Protocol vs. User Datagram Protocol 25

Chapter 3: Layered Data Replication **27**

3.1 Description 27

3.2 Comparison with Other Implementations 32

3.3 Implementation 34

3.3.1 Client 34

3.3.2 Directory Servers 38

3.3.3 Replica Servers 40

3.4 File Handling 42

3.5 Debugging 42

3.6 Encountered Problems 43

Chapter 4: Experimental Evaluation 45

4.1 Planetlab Platform 45

4.2 Scenarios 47

4.3 Results 50

4.4 Evaluation Conclusion 57

Chapter 5: Conclusion and Future Work 59

5.1 Summary 59

5.2 Future Work 60

References 62

TABLE OF FIGURES

Figure 1: Timeline of NFS protocols[21] **8**

Figure 2: The client-server architecture of NFS [21] **9**

Figure 3: The overall organization of AFS [22] **11**

Figure 4: The internal organization of a Virtue workstation [22] **12**

Figure 5: Side Effects in Coda's RPC2 system [22] **13**

Figure 6: Sending invalidation messages in concurrent [22] **13**

Figure 7: Connection-oriented socket [31] **24**

Figure 8: Connectionless socket [32] **25**

Figure 9: TCP segment structure [33] **26**

Figure 10: UDP segment structure [34] **27**

Figure 11: Clients Transitions [13] **30**

Figure 12: Replicas Transitions [13] **31**

Figure 13: Directories Transitions [13] **32**

Figure 14: Client Read Operation [13] **36**

Figure 15: Client Write Operation [13] **38**

Figure 16: Readers Operation Latency **50**

Figure 17: Writers Operation Latency **51**

Figure 18: Operation Latency with Different Replicas **52**

Figure 19: Operation Latency for Different Directory Choices **53**

Figure 20: Operation Latency while having different file sizes **54**

Figure 21: Average read operation latency using sequential and concurrent readers **56**

Figure 22: Average write operation latency using sequential and concurrent writers **57**

ACRONYMS

NFS – Network File System

VFS – Virtual File System

AFS – Andrew File System

RPC – Remote Procedure Call

xDR - eXternal Data Representation

UDP – User Datagram Protocol

RVID – Replicated Volume Identifier

VID – Volume Identifier

WORM – Write-Once Read-More

LDR – Layered Data Replication

TCP - Transmission Control Protocol

UDP - Universal Datagram Protocol

SMR - State Machine Replication

SFS – Secure File System

GFS – Google File System

BFT – Byzantine Fault-Tolerance

SSS - Shamir’s Secret Sharing

FAB - Federated Array of Bricks

CHAPTER 1

Introduction

Distributed file systems are divided in two types of inter-processor communication models: the shared-memory model and the message-passing model [1]. In the first, *n* processors communicate by writing and reading to/from shared registers, while in the second, *n* processors, equipped with private memory, are located at the nodes of a network and communicate by sending messages over communication links. In this thesis we developed an implementation that uses principles of both models in order to provide a solution that works in a reliable manner on an asynchronous platform, where dynamic changes on the nodes happen and fails may occur.

1.1 Motivation

In this thesis, we provide an implementation and experimental evaluation of algorithm LDR [13]. LDR has the potential of being the basis of a distributed file system that achieves the three main goals that any Distributed File System wish to achieve, which are replication, performance and consistency guaranties and all these without using any high level functions such as distributed locking [18], embed physical writes to the data within a logical read [1, 19] or group communication [21], that negatively affect performance [22].

Files are replicated in more than one replica servers to increase the performance and reliability of the system. In order to keep the system's consistency intact while multiple copies of a file are distributed in multiple servers, each server acts as an atomic register that always offers the most

up-to-date information to the clients. In addition to that the replication takes place in a transparent manner to the clients, who have no idea of the number of replications on the system.

Nodes are either clients or servers. Servers are either directories or replica servers. Directories share a global view of the replica servers on the system, while Replicas store the actual data of the system. The main factor that improves system's performance is that we focus more on doing cheap operations on the metadata information of a large file than on its actual content.

On the other hand, atomicity in the Directory Servers is achieved using two communication phases when a client reads the metadata of a file. After receiving the metadata it finds the max tag of a file and the most up-to-date replica list and re-sends them to all Directories. On Replica Servers, atomicity is achieved using a secure variable, and a reader can read a file's content if and only if its secure flag equals to one, and in the case that this file has been removed, the value of the file with the highest version and a secure level equal to one.

It is noteworthy, that in our implementation consensus is not wanted. It’s a factor unnecessary which only decreases the performance of the system. Instead of that we make use of a failure factor f, and accept $f+1$ responses or majority of responses from the servers, according to the phase that we are dealing with.

Most of the well-known distributed file systems provide replication in a way that affects either performance using composite calculations or consistency by making it lazy. The designers of LDR guaranty that none of these goals is affected. We wanted to test the algorithm through an evaluation on Planetlab [15] and find out the strengths of the algorithm and in what level their assumptions were right.

1.2 Contribution

The contribution of the thesis is the implementation and experimental evaluation of algorithm LDR [13]. In particular, the implementation supports the most common types of files, such as *.pdf, *.doc and *.txt files. It also supports *.jpg images.

In addition to that, communication is achieved using TCP communication sockets. Each time a client wishes to read or write, it creates a TCP socket with a server, which we may characterize as their communication session, and reads or write data to/from them in a reliable manner. In our implementation, exchanged messages are divided into control messages and actual data messages. Both of them have a maximum length of 944 bytes.

Also, every time an operation is completed, the client or replica server does a type of checksum test, by calculating the digest of the file using a well-known Hash Function[16], slightly modified. It then sends it back to the other as an acknowledgement who in turn compares it with the digest it produced for the same file, using the same method. We do that data integrity test as a first level check on the security of the implementation, because we haven't added any authentication check on the system. It is something that also offers us a technical check on the correctness of the file's data transfer when we divide it into blocks.

The experimental evaluation took place on the PlanetLab[15] platform. We tested a scenario with fixed numbers of replicas, directories and file size and different number of readers, increased by 10 every time we run the scenario. We noticed that without having any writers running in concurrent with the readers. This is because for every reader a new thread was created to service the reader's request, and with all the requests run in concurrent as one, the time needed for all readers was almost the same. In a second scenario, we took the results for different numbers of

replicas and all the other fixed, while in the third for different numbers of directory servers. In the final scenario we took the results for different sizes of files. From these scenarios we concluded that for a small number of clients, for example thirty (twenty readers and ten writers), no more than three replica servers and three directories servers are needed to serve them. Also when we tried a larger file size (500KB), a greater block size was needed (than 944bytes) because this size drove to a lot more packages exchanges, something that increased the time for all operations to complete, and a lot of connection timeouts from clients. Despite that, the atomic consistency remained intact. Concurrent read and writes always returned the latest version of a file.

1.3 Thesis Organization

Chapter 2 gives us a background on distributed file systems. First, we describe a basic Atomic Read/Write Registers protocol, we then give the descriptions of a variety of distributed file systems. Finally, we give a presentation on sockets and data transfer using TCP or UDP. Chapter 3 gives our implementation details. We start with a description of the Layered Data Replication algorithm and continue with a small comparison between LDR algorithm and the distributed systems described in Chapter 2. Then we present the three types of processes that we implemented, client, directory server and replica server. Next we give the details on file handling and debugging of the three programs. Chapter 4 gives the experimental evaluation details. Some information about Planetlab[15], the scenarios, and the results and conclusions are presented. Finally, in Chapter 5 we present our conclusion and we give recommendations for potential future work.

CHAPTER 2

Background

In this chapter we first present a basic Atomic R/W Registers protocol. We next move on the descriptions of the distributed file systems that reviewed. We also go through sockets and data transfer protocols.

2.1 Atomic R/W Registers

Designing of fault-tolerant algorithms is easier in the shared-memory model because the shared memory helps the processors to obtain a more global view of the system. The ABD protocol [1] supports porting of shared-memory algorithms to message-passing systems. It adverse serves failures, as a conceptual basis for several storage systems, and for universal service implementations, for example using the State Machine Replication (SMR) [7,8].

ABD relies on atomic operations (Linearizability [2]). An atomic, single-writer multi-reader register is an abstract data structure. Each register is accessed by two operations, *writew (Value)* that is executed only by some specific processor *w*, called the writer, and *readr (Value)* that may be executed by any processor r, *1<=r<=n*, called a reader. An operation precedes another if it returns before the other operation is called. Two operations are concurrent if neither of them precedes the other.

The values returned by these operations, when applied to the same register, must satisfy the following two properties:

- Every read operation returns either the value written by the most recent preceding write operation (the initial value if there is no such write) or a value written by a write operation that is concurrent with this read operation.

- If a read operation R1 reads a value from a write operation W1, and a read operation R2 reads a value from a write operation W2 and R1 precedes R2, then W2 doesn't precedes W1.

The combination of failures and asynchrony drives to possibility of system partition if more than a majority of the nodes fail. In this case two operations may proceed without knowing each other's existence, something that may break atomicity if a read operation misses the value of a preceding write operation.

One reason the ABD protocol is well-known is due to its simplicity, at least in the unbounded version. In this version, there is one simple system with one writer and x readers. All n nodes store a copy of the current value of the register. Each value is associated with an integer donate version#.

To write a value, the writer sends a message *write(value,version#)*, with the new value and an incremented version number, to all nodes and waits for $n-f$ acknowledgments, where f is a failure factor and $n-f > n/2$. To read a value, a reader queries all nodes, and after receiving at least $n-f$ responses, picks the value with the highest version number. To solve the problem where two non-overlapping reads both overlapping a write, with the risk of obtaining out-of-order values, ABD sets the reader to write back the value it is going to return. In this way it ensures atomicity of reads.

The bounded version of ABD [1] includes bounding the version numbers and replacing the communication with the majority of nodes with a quorum. The key to bounding the version numbers is to know which of them are currently in use in the system. Tracking the version numbers is done by having a reader "record" a version number before forwarding it with a value. A quorum system is a collection of subsets of nodes, with the property that each pair of sets has at least one common node, so to have a nonempty intersection. Each operation must communicate with a quorum.

Lynch and Shvartman presented an extension of the ABD protocol for MWMR atomic objects [4]. In this extension reading takes two rounds exactly like in ABD, except that the tag equals to pair (num, writer id). Writing now needs two communication phases. A writer must first calculate the maxTag and then

announce the pair (maxTag++,value) to the servers. This protocol took the nickname RAMBO.

To make the emulation more robust, in dynamic systems, reconfiguration must take place every time a change happens in the system. DynaStore [3], RAMBO [4] and Paxos [5] use reconfiguration in order to deal with the dynamic node participation (nodes may come and go arbitrary). In Rambo, a new configuration can be proposed by any process, and once it is installed it becomes the current configuration. In DynaStore, processes suggest changes and not configurations, and thus, the current configuration is determined by the set of all changes proposed by complete reconfigurations. For example, if a process suggests to add p1 and to remove p2, while another process concurrently suggests adding p3, DynaStore will install a configuration including both p1 and p3 without p2. In both algorithms, a non-faulty quorum is required from the current configuration. This suggests that consensus may not be needed for dynamic storage. Paxos, implements the SMR [7,8], and allows one to dynamically reconfigure the system by keeping the configuration itself as part of the state stored by the state machine. DynaDisk [9] is a data-centric reconfiguration system based on DynaStore. It allows clients to decentralized add or remove storage devices, without stopping ongoing read/write operations.

2.2 Other Distributed File Systems

This subchapter provides an overview of the distributed file systems from the literature.

2.2.1 Network File System

The NFS system [20, 21] is a distributed file system originally developed by Sun Microsystems in 1984, for transparent remote access to shared files across networks. It is designed to be portable across different machines, operating systems, network architectures, and transport protocols.

Sun used version one only for in-house experimental purposes. NFS version two (NFSv2) is the older from the ones used outside Sun and is widely supported. NFS version 3 (NFSv3) has more features, including 64bit file handles, Safe Async writes and more robust error handling. NFS version 4 (NFSv4) works through firewalls and on the Internet, no longer requires portmapper, supports ACLs, and utilizes stateful operations. Today, NFS exists as version 4.1, which adds protocol support for concurrent access across distributed servers (called the *pNFS extension*). The timeline of NFS, including the specific RFCs that document its behavior, is shown in Figure 1.

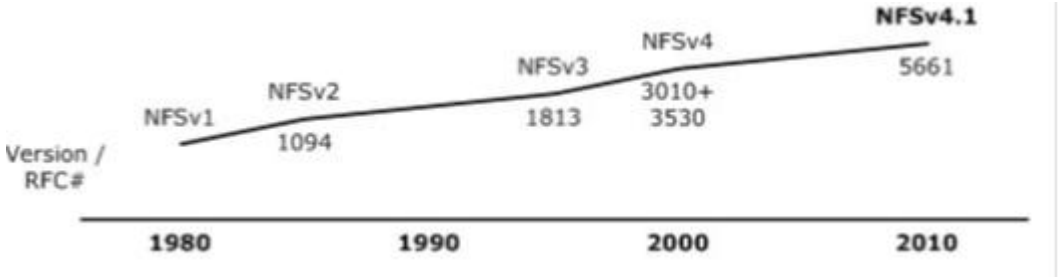


Figure 1. Timeline of NFS protocols [21]

NFS follows the client-server model of computing (see Figure 2). The server implements the shared file system and storage to which clients attach. The clients implement the user interface to the shared file system, mounted within the client's local file space.

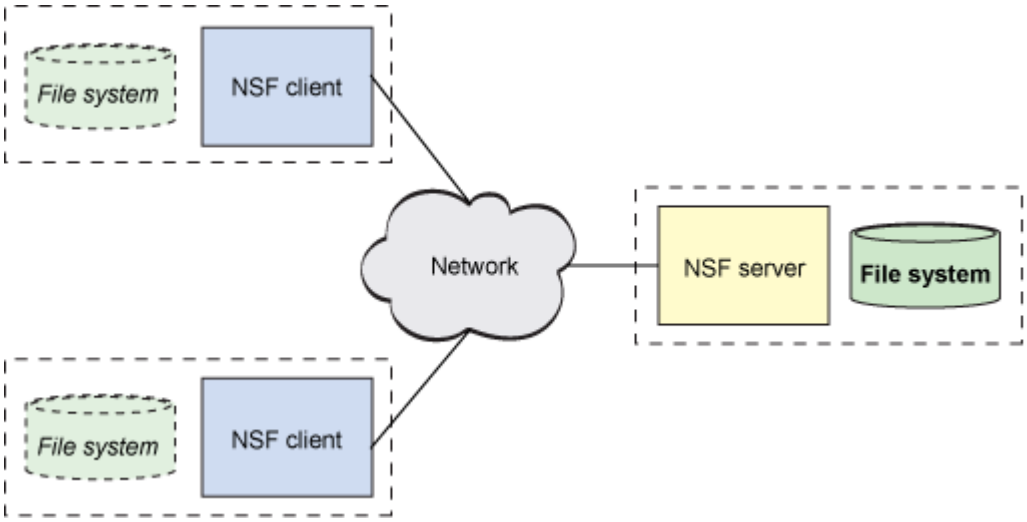


Figure 2. The client-server architecture of NFS [21]

Within Linux, the virtual file system switch (VFS) provides the means to support multiple file systems concurrently on a host. Once a request is found to be destined for NFS, VFS passes it to the NFS instance l. NFS interprets the I/O request and translates it into an NFS procedure (`OPEN`, `ACCESS`, `CREATE`, `READ`, `CLOSE`, `REMOVE`, and so on). Once a procedure is selected from the I/O request, it is performed within the remote procedure call (RPC) layer. As the name implies, RPC provides the means to perform procedure calls between systems.

RPC also includes an important interoperability layer called *external data representation* (XDR). XDR takes care of converting types to the common representation (XDR) so that all architectures can inter-operate and share file systems.

The request is then transferred over the network given a transport layer protocol. Early NFS used the Universal Datagram Protocol (UDP), but today TCP is commonly used for greater reliability.

From the client's perspective, the first operation to occur within NFS is called a *mount.* Mount represents the mounting of a remote file system into the local file system space.

NFS uses a TTL (time to live) based approach at the client-side to invalidate caches. It is also stateless. For that reason the server does not maintain any locks between requests and a write may cover several RPC requests (mixed file versions). When it comes to consistency, it is not always guaranteed, and we can characterize it as weak. The latest data will still not be available to another client sharing the file until the TTL period is over. The design of NFS involved simplicity, and the assumption that clients will not need to do any concurrent updates.

2.2.2 Coda File System

The Coda File System [22] was developed at Carnegie Mellon University (CMU) the 1990, has now been incorporated into a number of popular operating systems based on UNIX, like Linux. The Coda differs from NFS in many ways especially when it comes to the goal for high availability. This goal led to advanced methods of use of the cache, which allows a client to continue an operation while disconnected from the server.

It has been designed to be a scalable secure and highly available distributed file system. Coda is a descendant of version two of Andrew file System (AFS) [8]. The AFS was designed to support the entire CMU community, which means that about 10000 workstations would have access to the system. To meet this requirement, AFS nodes are divided into two groups. The first group consists of a small number of file servers named Vice, which are subject to centralized management. The other group consists of a larger collection of workstations named Virtue which give access to users and processes to the file system as shown in Figure 3.

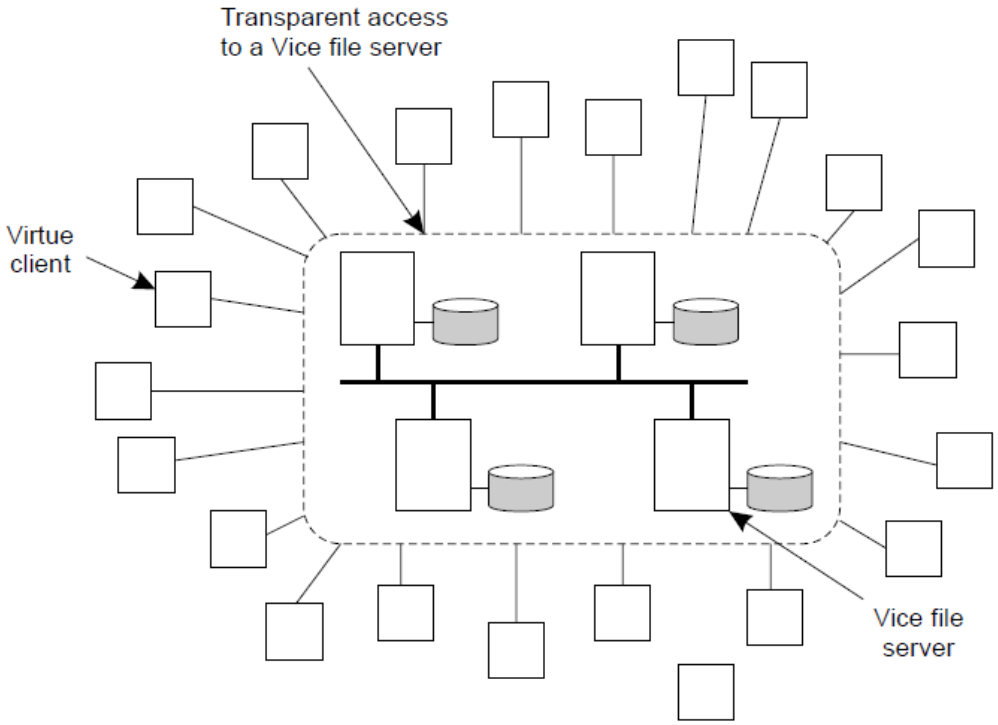


Figure 3. The overall organization of AFS [22]

Coda follows AFS's organization. Each workstation named Virtue is hosting a user-level process called Venus, whose role is similar to that of an NFS client. Venus is a process responsible for providing access to records kept by Vice file servers. Besides that, in Coda, Venus is also responsible for allowing clients to continue an operation, even if the access to the file server is (temporarily) impossible. This additional role is a big difference from the approach followed in the NFS.

Figure 4 shows the internal architecture of a Virtue workstation. The important thing is that Venus runs as a user-level process. And in this case there is a separate level of a VFS that takes all the calls from client applications and forwards them either to the local file system or at Venus.

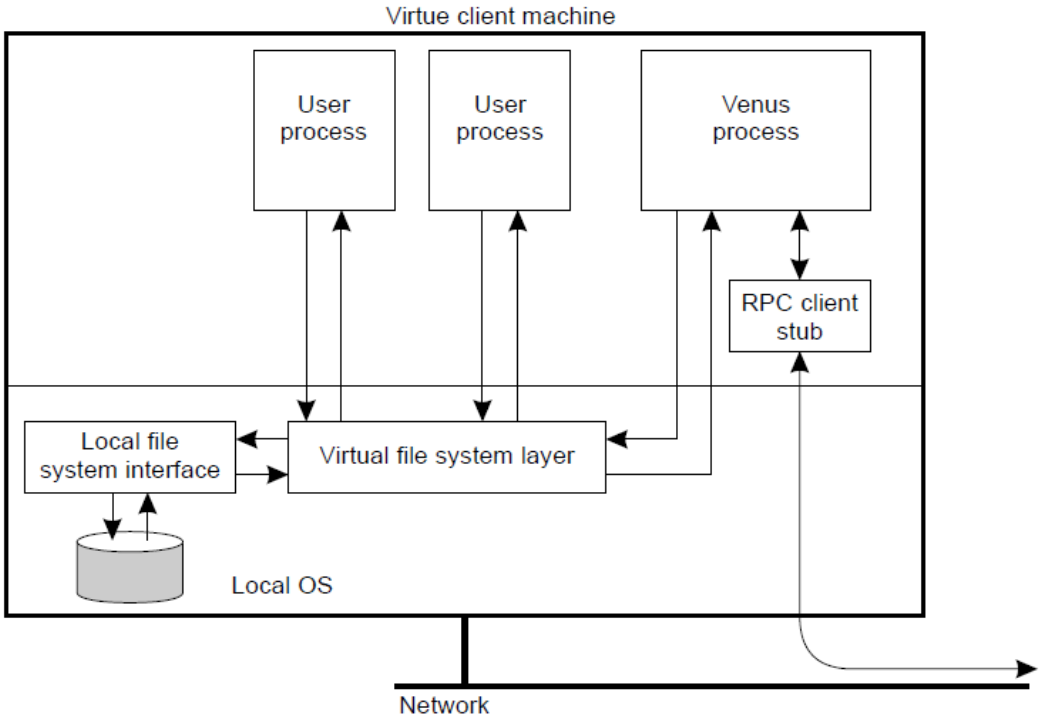


Figure 4. The internal organization of a Virtue workstation [22]

The process communication in Coda is performed using RPC2, which is much more complex than traditional RPC. RPC2 provides reliable RPC over the unreliable UDP. Each time a remote process is called, client RPC2 code starts a new thread, which sends a call request to the server and then is blocked until it receives a response. Because the time required to complete the processing of the application is unspecified, the server sends messages to the client, at regular periods of time, to inform him that it still works on its request.

An interesting feature of RPC2 is the support for side effects (Figure 5). A side effect is a mechanism by which the client and server can communicate using a protocol that depends on the application. In this way (connection), on time image data transfer to the client is achieved.

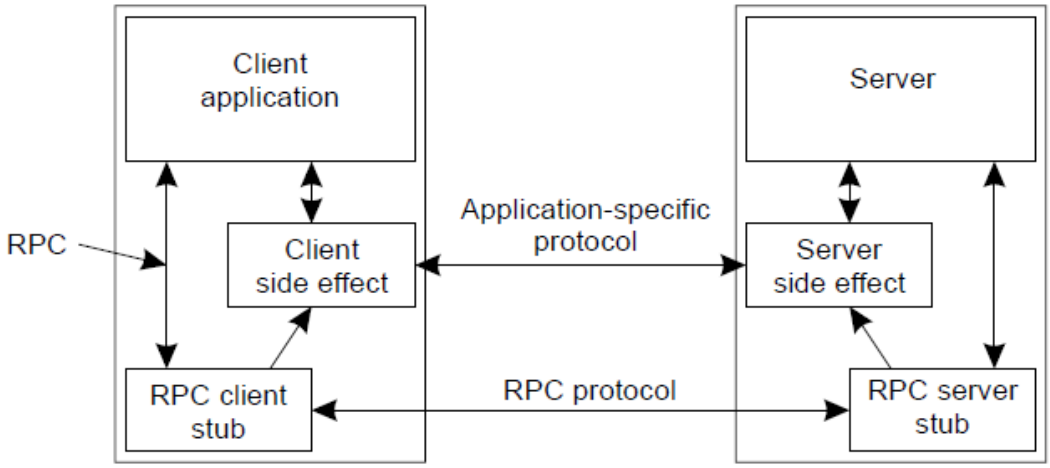


Figure 5. Side Effects in Coda's RPC2 system [22]

An important issue in Coda design is that servers keep track of which clients have a local copy of a file. When the file is modified, the server cancels the local copies by alerting clients via an RPC. It sends an invalidation message to all clients in concurrent (Figure 6). Also during the usual expiration time it realizes that some clients may not respond to the RPC call, and can conclude that these clients have collapsed.

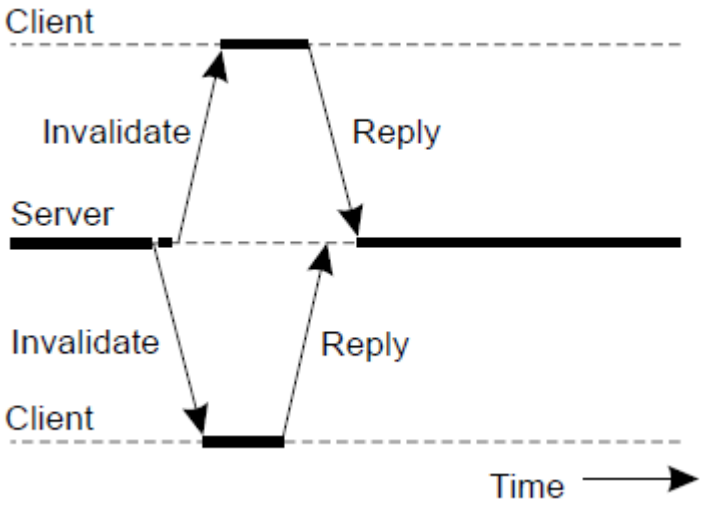


Figure 6. Sending invalidation messages in concurrent [22]

Concurrent RPCs are implemented through the MultiRPC system [9] and is transparent to the called party that does not distinguish them from a normal RPC.

Coda wants to address the problem of temporary unavailability, which may be due to failures of the network, of servers or any mobile client, by interpreting each session as a transaction. When a client starts a session, all data associated with this session is copied to the client machine, including the version number of each data item. If a network failure appears, Venus process will allow the client and the server to continue and complete the execution of the session(s) as if nothing happened. Later when the connection to the server is restored, the updates will be transferred there in the same order made to the client. Server accepts an update only if it leads to the next file version.

Coda's consistency is neither atomic nor sequential. It does not allow inconsistencies at any point of time, but updates take place when a server is aware of changes. This is due to the fact that Coda has made an assumption that sequential write sharing between users is relatively rare in UNIX environments, so conflicting updates are likely to be rare.

2.2.3 Plan 9

Plan 9 [23] is based on the idea of having a few central servers and multiple client machines, where servers are powerful and relatively inexpensive computers that use microcomputers technology, with their management being centralized. On the other hand, client machines are simple and have only a few tasks to do.

The idea for the system captured largely by the same group of people who were responsible for the development of UNIX at Bell Laboratories, and began the decade of 1980.

Plan 9 is not that much of a distributed file system, but rather a distributed system based on files. All resources are accessed in the same way, and each server provides a hierarchical namespace to the resources it controls. A client can mount a local namespace provided by a server, creating its own private namespace like NFS.

Although there is a distinction between clients and servers in the description, such a distinction is not always clear in plan 9. Here servers often act as clients of other machines while customers can export their resources to servers.

Network communication is carried out through a local protocol with the name 9P. The 9P runs over a reliable transport protocol. For local networks, it uses the Internet Link (IL), while for wide area networks the TCP.

Network connections are represented by a file system, which consists of a collection of specific files. For example, a TCP connection consists of files ctl, data, listen, local, remote, status. ctl is used to send files to control connection, data for the exchange of data by simply performing the read and write functions, listen for pending applications for connection setup, local gives information about caller connection side, remote gives information about the other side of connection, and status gives information about the current status of the connection.

Plan 9 has several servers, each of which implements a hierarchical namespace. Let's take for example the file server, which is a standalone system that runs from a dedicated machine. From a logical standpoint is organized as a storage system of three levels. The lower unit consists of a Write-Once, Read-Many (WORM), which provides mass storage. The intermediate level consists of a collection of magnetic disks acting as one large system cache for the WORM device. When a file is accessed, is read from WORM device and stored in the disk. Also any changes are temporarily stored in the disc. Once a day, all changes

are written to WORM device, which provides in this way an incremental backup of the entire file system on a daily basis. The higher level consists of a large collection of small area buffers, which operate as a cache for the magnetic disks inside the main memory.

Plan 9 implements UNIX file-sharing semantics, allowing the file server to always keep a copy of a file. All update functions are always promoted to the server. The functions of simultaneous clients are handled in an order specified by the server, but the updates are not lost ever.

Also, it provides a minimum of support for using caching and replication. To avoid unnecessary file transfers, clients can use locally stored copies, given they are valid. The validity of a file in the cache is checked by comparing the version number of this file to the one on the server. If the numbers differ, the client cancels the data in its cache and takes a new copy from the server.

Plan 9 provides a more unified approach to the backup problem by implementing a snapshot feature. A snapshot is a consistent read-only view of the file system at some point in the past. The snapshot retains the file system permissions and can be accessed with standard tools (ls, cat, cp, grep, diff) without special privileges or assistance from an administrator.

To address problems of consistency and portability among applications, Plan 9 uses a fixed color map, called rgbv, on 8–bit–per–pixel displays. Although this avoids problems caused by multiplexing color maps between applications, it requires that the color map chosen be suitable for most purposes and usable for all.

Plan 9 also uses Cfs. Cfs is a user–level file server that caches data from remote files onto a local disk. It is normally started by the kernel at boot time, though users may start it manually. On each open of a file cfs checks the consistency of cached information and discards any old information for that file.

2.2.4 xFS: File System without Server

The xFS file system [24] (not to be confused with XFS [25]), is an unusual distributed file system because its design does not include servers. The entire file system is spread over several machines, among who are clients. It was designed for operation in a local network where machines are connected via high-speed connections.

In its architecture there are three different types of processes: the storage servers, the metadata managers and the clients. The storage server is a process responsible for storing portions of a file. Together, the storage servers implement a correlation of virtual disks similar to that of RAID [26]. Metadata managers are processes responsible for monitoring the position where the data segments of a file are stored in reality. These segments can be spread across multiple storage servers, so the manager forwards requests from clients to the appropriate storage server. Finally, a client is a process that accepts user requests to perform operations on files. Every client has the ability to store in the cache, and can provide other clients with locally stored data. A key design principle of xFS is that any machine can take on the role of the client, the manager, or the server.

Communication is achieved using active messages. In an active message the operator is determined on the side of the recipient along with the necessary parameters for a call. When the message arrives, the operator is immediately called and executed. No other messages can be delivered during the execution of the operator. The main advantage of this approach is its efficiency. However, active messages complicate communication design. For example, operators are not allowed to be blocked. Also, they must be relatively short, because their execution doesn't allow further network communication with operator's computer.

When it comes to file consistency, xFS manages to provide sequential and not linear consistency.

Sequential Consistency (SC) produces lock contention, something that automatically gives poor performance. In SC, all readers see a write operation in the same way, and on order to allow concurrent operations, the system uses locking mechanisms.

2.2.5 Federated Array of Bricks

The Federated Array of Bricks [27] is a low-cost alternative solution to disk arrays for enterprise-class storage systems. It is a logical disk system that provides reliability and performance and its design makes it scalable from very small to very large systems. This is achieved by adding together storage bricks that are actually a module consist of disks, a CPU, NVRAM, and network cards. A disk block is stored in multiple bricks, and redundant paths are created between all components of the system, in order to overcome brick failures.

Client systems connect to FAB bricks using standard protocols such as Fibre Channe l[12] or iSCSI [13]. Bricks are connected to each other using local area networks like Ethernet. FAB presents to clients a number of logical volumes. Because of the fact that FAB is decentralizes a client can ask from any brick to create, re-size, or access a logical volume.

FAB breaks volumes into fixed-size segments, each containing a number of blocks. The default segment size is 8GB and the default block size 1KB. A number of segments are gathered into groups, in order to enable efficient metadata management. Segments are used in layout management and groups for replication and availability.

Each brick runs three software modules, the *coordinator*, the *block-management* and the *configuration-management*. Coordinator receives clients' requests and coordinates disk read and write requests on

behalf of clients. Block-management reads and writes disk blocks. And configuration-management uses Paxos distributed consensus algorithm [5] to replicate configuration information.

FAB it is based on RAMBO [4]. When writing, the coordinator generates a new timestamp and writes the value and timestamp to the majority of replicas. And when reading, it reads from the majority and returns the value with the newest timestamp.

FAB manages to implement atomic consistency. Because coordinator may fail, to prevent leaving a new value to a sub-majority of the replicas something that may affect linearizability, each replica of a logical block keeps two timestamps, the timestamp of the value currently stored and the timestamp of the newest on-going write request. To ensure linearizability a write operation has two phases, while read has one in a normal scenario and three if timestamp indicates a past failure.

When it comes to the load, it is uniformly distributed over the bricks. When new bricks join FAB, then it reassigns segment replicas from heavily loader bricks to the new bricks. The FAB replica-management protocol permits actual data reads to be made from any replica, while the other replicas only provide timestamp information.

2.2.6 Google File System

The Google File System (GFS) [28] is a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance using inexpensive machines, and having high performance while working with a large amount of clients.

Its architecture consists of a single master and multiple chunkservers. Each chunk has a 64bit chunk

handle assigned to it by the master, the time of its creation. For reliability, each chunk is replicated on multiple chunkservers, whose job is to store them on local disks. A chunk's size has been chosen to be 64MB, which is much larger than typical file systems.

The master maintains all file system metadata, which include namespace, information about access control, mapping from files to chunks, the current locations of the chunks, and also controls activities such as chunk lease management and garbage collection of orphan chunks. Periodically the master communicates with each chunkserver using Heart Beat messages in order to give them instructions and get their stage.

Clients communicate with the master for the metadata information and then with the chunkservers for the operations on the real data. Clients don't cache file data due to their size. They cache only the information about the chunkservers given by the master, but only for a small amount of time. As for chunkservers they don't need to cache file data because chunks are stored as local files.

An important issue in GFS is the operation log. Namespaces and file-to-chunk mapping are also stored in the operation log on the master's local disk. It holds all metadata changes. By that it allows us to update the master state simply, reliably, and without risking inconsistencies in case it crashed. It also serves as a logical time that defines the order of concurrent operations. Files, chunks and versions are all uniquely identified by the time they were created.

The mutations (changes of metadata or real content) on a chunk are applied to all replicas, in a general order specified by the master and within a lease by the serial numbers (serial order) assigned by the primary replica. If the primary wishes to extent the lease (it lasts 60 secs), because the mutation is not finished, the primary can do that by requesting extra time within the Heart Beat message, it exchanges with the master.

GFS implements sequential consistency. A simple scenario that gives the general idea of GFS when having for example a write operation is the follow: The client communicates with the master to ask for the chunkservers that hold the lease and the locations of the other replicas. If the master has no primaries it chooses one. The master replies with the information. The client caches the data for future mutations. It searches the locations of the other replicas and sends them the data. They will store them in an internal LRU buffer, until they are used or out of date. They will send acknowledgments back to the client. When it receives acks from all, it will send a write request to the primary. The primary will assign serial numbers to all the mutations it receives from multiple clients, for serialization. The primary, then forwards the write request to all the secondaries, and they apply the mutations in the same serial order. When they complete the operation they reply telling it to the primary. Finally the primary reply back to the client. If the primary reports any errors to the client, then its request is considered as failed. If the amount of data is large, the write operation breaks into multiple others.

Data flow is distinguished from control flow. Control flows from the client to the primary, and then to all secondaries. On the other hand, data flows linearly along a chosen list of chunkservers following a pipeline style.

GFS also provides an atomic operation called record append, and it is used when many clients want to append the same file concurrently, without the use of some lock manager. It does not guarantees that all replicas are bytewise identical, but it guarantees that the data is written at least once as an atomic unit.

2.2.7 Belisarius

Belisarius [29] is a Byzantine Fault-Tolerance system that also provides confidentiality protection through Shamir's secret sharing (SSS) algorithm [30], using an economical and simple architecture, having a minimum performance impact by moving a significant part of the protocol to the client, and without requiring any complex key management system.

In the system there is an arbitrary number of client nodes and a fixed number *n* of server nodes. Nodes can be correct or faulty. Client nodes can be authorized (trusted) or unauthorized. Correct server nodes are responsible to enforce access control of all clients. The system has a correct operation if only *f* servers are Byzantine in a set of *2f+2* servers. If no Byzantine shares appear, then the client needs only *f+2* shares to compute the secret. If there is one Byzantine share then the client may have to wait for *2f+2* replies.

To make a secret S, *n* shares are created, and *t* of them are needed to reassemble to secret. In order to ensure that at least one share that comes from a correct server *t* must be greater than *f*. The verification of the shares is achieved using quorums.

The three main components of Belisarius are: the client-side confidentiality handler, the BFT communication protocol, and the server-side transparent manipulation if obfuscated data. Client breaks a value into shares using SSS, and one write(key, value) operation breaks into n encrypted write(key, share) operations. Then the client broadcasts them to each server. The key is a server-specific session key. For each REQUEST message from the client, there is a PRE_PREPARE message from the primary replica, who is responsible for dictating the total order of requests, one PREPARE message from each backup replica, and a COMMIT message from each replica. For confidentiality reasons no single server contains any usable data by itself. The system assures strong consistency for transactions.

2.3 Sockets

Sockets are the sending and receiving points for message delivery. When the delivery is achieved using TCP, the sockets are connection-oriented and are called Stream Sockets (see Figure 7). There is also the connectionless type of sockets; the ones that use UDP, and are called Datagram Sockets (see Figure 8).

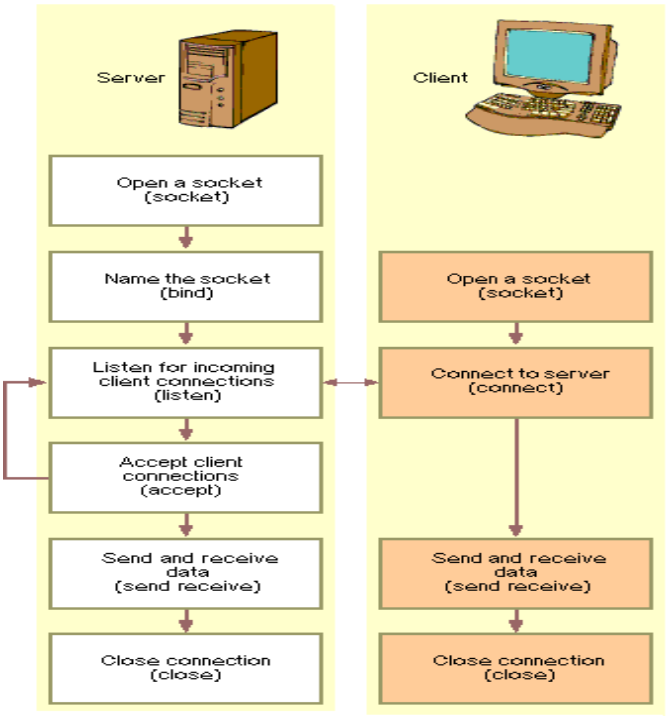


Figure 7. Connection-oriented socket [31]

In a connection-oriented socket, the process followed by server is firstly the creation of a socket. Next the socket is bind to an IP address and a port, and then it listens for connections. When it receives one, the socket accepts the connection, and goes on a loop, where it received data from the other endpoint and replies by sending its respond. When the data transfer is completed, the server closes the connection and unlinks the socket. On the client side, we also have the socket creation, and then the client attempts to

establish a connection with the server. If the connection is accepted by the server, the client starts sending its data, and receives the server responds. This may be repeated until all data packets are sent and received from both sides. When the data transfer is finished the client closes the socket.

As for connectionless socket, we have both sides creating a socket. The server binds the socket to an IP address and a port, but does not listen or accept connections. It just receives and sends data packets from and to clients. On the client side, no connection is attempted to be established with the server. The client after socket creation, it just send data packets to the server and receives its responses. Because UDP doesn't have a pipe, when a process wants to send a set of bytes to another process, the sending process must attach the destination's process address to the set of bytes. And this must be done for each set of bytes the sending process sends. When the client finishes with the data transfer it closes its socket.

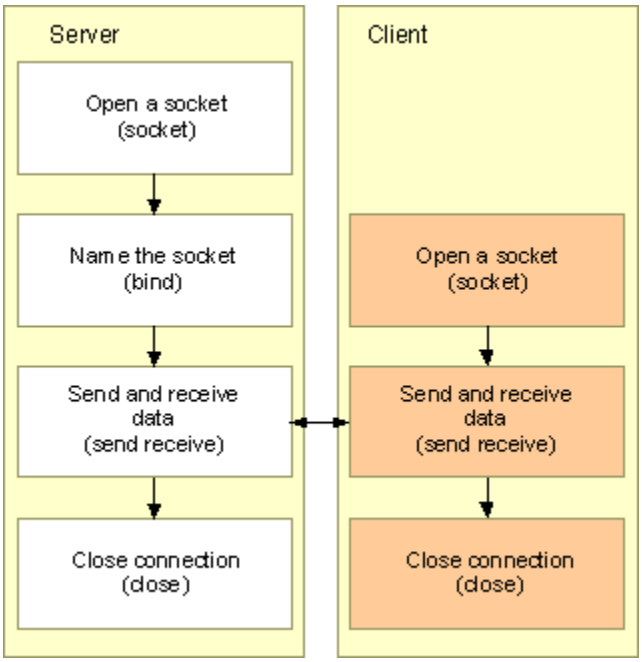


Figure 8. Connectionless socket [32]

2.5 Transport Layer Protocol vs. User Datagram Protocol

There are two types of protocols for data transfer through the Internet, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). In our implementation we have used the TCP.

TCP (Figure 9) is connection-oriented, which means that the two processes must execute a three-way handshake first to establish the parameters of the ensuing data transfer. Due to the fact that this protocol runs only at the end systems, the intermediate network elements don't maintain TCP. The applications that use it are those that are not time critical (HTTP, HTTPs, FTP, SMTP, Telnet etc), because TCP is a slow protocol, and the reason for that is that in its effort to provide reliable transfer it rearrange the data packets in order according to their sequence number, it takes acknowledgments and executes error checking. But TCP is not chosen for its speed, but for its reliability. When data transfer is been done using this protocol, there is the absolute guarantee that the data transferred remain intact and arrive in the same order in which they were sent. It can also handle congestion control, and does Flow Control.

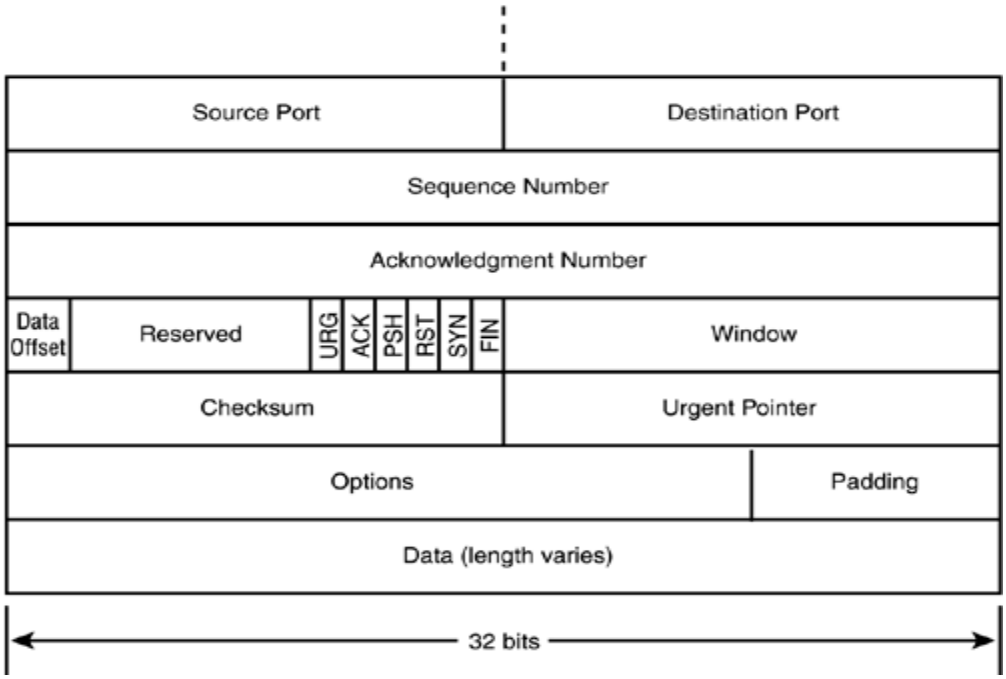


Figure 9. TCP segment structure [33]

On the other hand, UDP is a connectionless protocol; no handshake is needed for an endpoint to send data to another. It is faster than TCP, because there is no error-checking for packets or package sequence. That's why it is used for games, videos or applications that require fast transmission and are not affecting from packet lost because UDP is not a reliable transfer protocol but a best-effort protocol. Its stateless nature it is also useful for servers that answer small queries from huge numbers of clients, for example DNS, DHCP, TFTP, SNMP, RIP, VOID etc. Packets in UDP are also independent from each other and may be delivered out of order. If order is required this can be arrange by the application layer. As for its header size UDP is only 8 bytes (see Figure 10), 12 bytes less than TCP. Like TCP it provides Check Sum to each packet but no error-checking. Also it offers no congestion or flow control.

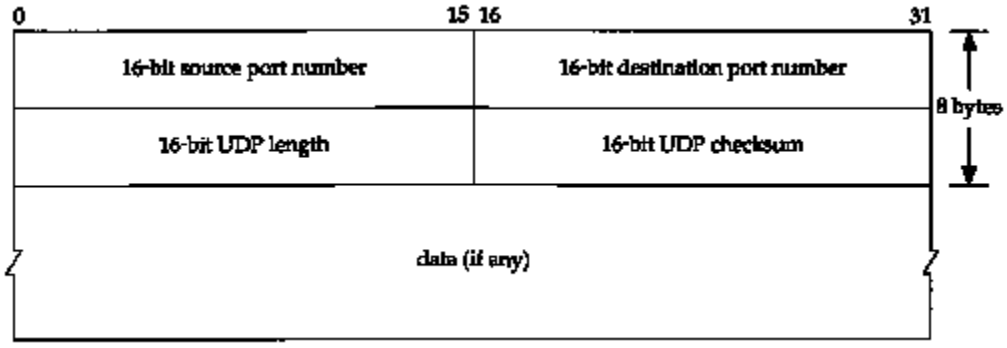


Figure 10. UDP segment structure [34]

CHAPTER 3

Layered Data Replication

In this chapter we give the implementation details. We start with a description of the Layered Data Replication algorithm and continue with a brief comparison between LDR algorithm and the distributed systems described in Chapter 2. Then we present the three types of processes that we implemented, client, directory server and replica server. Next we give the details on file handling and debugging for the three programs.

3.1 Description

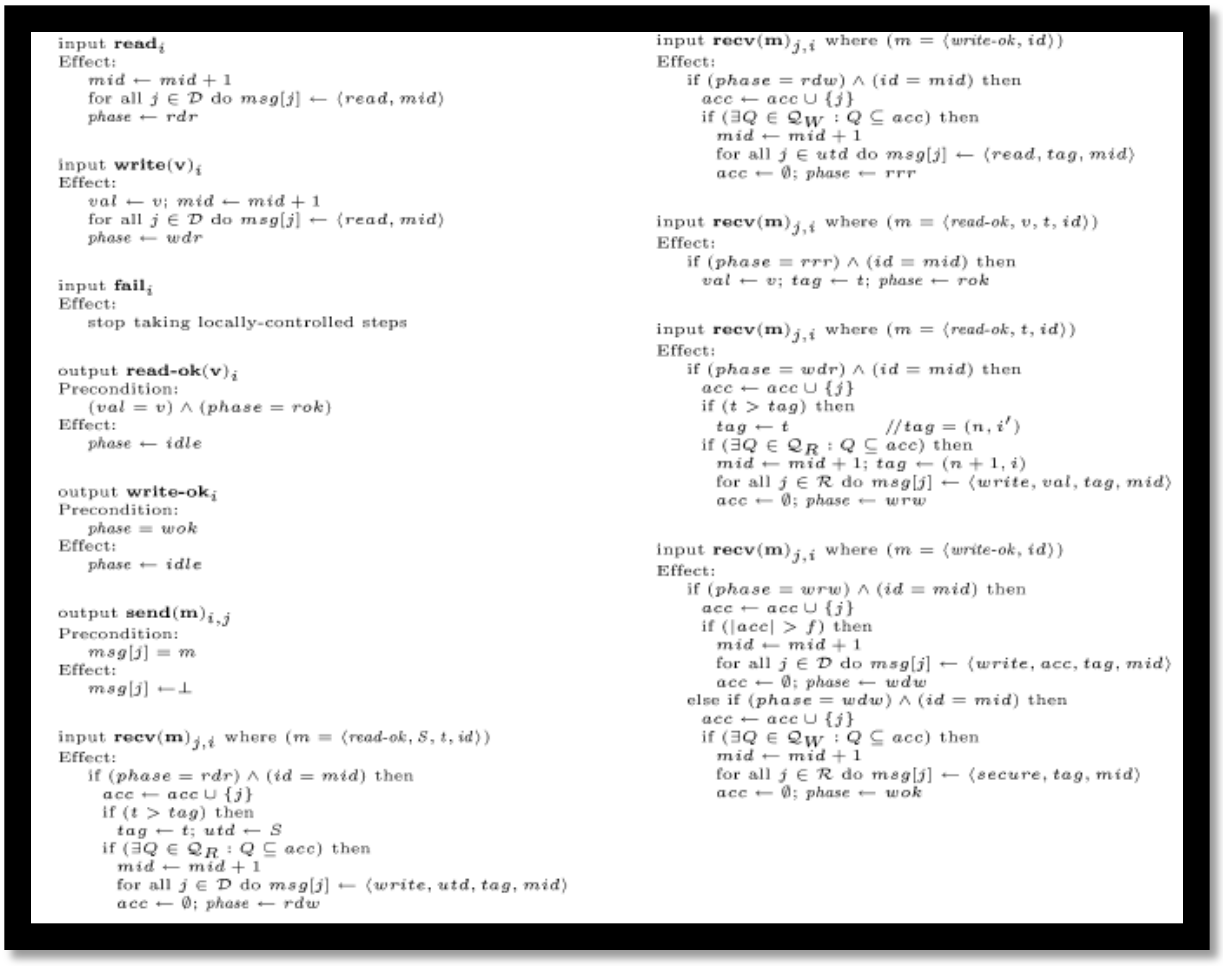
LDR runs on top of any reliable, asynchronous message passing network and it is suitable for both LAN and WAN settings, because according to [13], it tolerates high latency and network instability. The motivation behind the implementation of LDR algorithm is the fact that it manages to achieve the three main goals that any Distributed File System wish to achieve, which are replication, performance and consistency.

It is called LDR, because it has two layers of servers, the Directory Servers and the Replica Servers. The first one, stores the set of up-to-date Replicas and the latest tag, while Replica stores the real data. This is the main reason of how the algorithm deals with the performance penalty of data replication. It takes advantage of the fact that metadata are lighter than the size of the objects being replicated, and does more operations on them than in real data.

The implementation consists of n clients, a set R of Replica Servers and a set D of Directory Servers. Each of the client, replica, directory, has its own state variables and runs a different protocol than the other. The client takes an external input, and then connects with the directories and some replicas in order to perform the requested operation. Its state is described by three variables. Variable phase that takes a value according to the operation that takes place at a time. Variable $utd \in 2^R$, stores the set of replicas that the client thinks are the most up-to-date. And the mid is a message counter.

A Replica Server has one state variable the data of the form $\subseteq V \times T \times \{0,1\}$. V is the set of values, T is the set of *tags (version, writer id),* and the third triple goes for secure or not. A Directory Server has a $utd \subseteq R$ variable, and a $tag \in T$. The utd variable has the set of the latest up-to-date replicas, while tag is the tag associated with the value.

When it comes to the protocol, a client goes through four phases during the read operation (see Figure 11). The rdr, rdw, rrr and rok. During rdr, a client gets (utd, tag) from a quorum of directories. During rdw, the client writes (utd, tag) to a write quorum of directories. During rrr, the client reads the value of x from a replica in the utd set. This value must be associated with the tag. Read finishes with rok.



input $\mathbf{read}_i$
Effect:
$mid \leftarrow mid + 1$
for all $j \in \mathcal{D}$ do $msg[j] \leftarrow \langle read, mid \rangle$
$phase \leftarrow rdr$

input $\mathbf{write(v)}_i$
Effect:
$val \leftarrow v$; $mid \leftarrow mid + 1$
for all $j \in \mathcal{D}$ do $msg[j] \leftarrow \langle read, mid \rangle$
$phase \leftarrow wdr$

input $\mathbf{fail}_i$
Effect:
stop taking locally-controlled steps

output $\mathbf{read\text{-}ok(v)}_i$
Precondition:
$(val = v) \wedge (phase = rok)$
Effect:
$phase \leftarrow idle$

output $\mathbf{write\text{-}ok}_i$
Precondition:
$phase = wok$
Effect:
$phase \leftarrow idle$

output $\mathbf{send(m)}_{i,j}$
Precondition:
$msg[j] = m$
Effect:
$msg[j] \leftarrow \perp$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle read\text{-}ok, S, t, id \rangle)$
Effect:
if $(phase = rdr) \wedge (id = mid)$ then
$acc \leftarrow acc \cup \{j\}$
if $(t > tag)$ then
$tag \leftarrow t$; $utd \leftarrow S$
if $(\exists Q \in \mathcal{Q}_R : Q \subseteq acc)$ then
$mid \leftarrow mid + 1$
for all $j \in \mathcal{D}$ do $msg[j] \leftarrow \langle write, utd, tag, mid \rangle$
$acc \leftarrow \emptyset$; $phase \leftarrow rdw$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle write\text{-}ok, id \rangle)$
Effect:
if $(phase = rdw) \wedge (id = mid)$ then
$acc \leftarrow acc \cup \{j\}$
if $(\exists Q \in \mathcal{Q}_W : Q \subseteq acc)$ then
$mid \leftarrow mid + 1$
for all $j \in utd$ do $msg[j] \leftarrow \langle read, tag, mid \rangle$
$acc \leftarrow \emptyset$; $phase \leftarrow rrr$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle read\text{-}ok, v, t, id \rangle)$
Effect:
if $(phase = rrr) \wedge (id = mid)$ then
$val \leftarrow v$; $tag \leftarrow t$; $phase \leftarrow rok$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle read\text{-}ok, t, id \rangle)$
Effect:
if $(phase = wdr) \wedge (id = mid)$ then
$acc \leftarrow acc \cup \{j\}$
if $(t > tag)$ then
$tag \leftarrow t$ //$tag = (n, i')$
if $(\exists Q \in \mathcal{Q}_R : Q \subseteq acc)$ then
$mid \leftarrow mid + 1$; $tag \leftarrow (n + 1, i)$
for all $j \in \mathcal{R}$ do $msg[j] \leftarrow \langle write, val, tag, mid \rangle$
$acc \leftarrow \emptyset$; $phase \leftarrow wrw$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle write\text{-}ok, id \rangle)$
Effect:
if $(phase = wrw) \wedge (id = mid)$ then
$acc \leftarrow acc \cup \{j\}$
if $(|acc| > f)$ then
$mid \leftarrow mid + 1$
for all $j \in \mathcal{D}$ do $msg[j] \leftarrow \langle write, acc, tag, mid \rangle$
$acc \leftarrow \emptyset$; $phase \leftarrow wdw$
else if $(phase = wdw) \wedge (id = mid)$ then
$acc \leftarrow acc \cup \{j\}$
if $(\exists Q \in \mathcal{Q}_W : Q \subseteq acc)$ then
$mid \leftarrow mid + 1$
for all $j \in \mathcal{R}$ do $msg[j] \leftarrow \langle secure, tag, mid \rangle$
$acc \leftarrow \emptyset$; $phase \leftarrow wok$

Figure 11. Clients Transitions [13]

Writing also goes through four phases, the wdr, wrw, wdw and wok (see Figure 11). During wdr, a client reads (utd, tag) from a quorum of directories, and sets tag to be the highest than the largest tag it read. During wrd, the client writes (v, tag) to a set acc of replicas, where |acc| ≥ f +1, and f is the maximum number of allowed replica fails and must be less than |R|. During wdw, the client writes (acc, tag) to a quorum of directories to inform them for the most up-to-date replicas and the newest tag. After that, the client send a secure message to each replica to tell them that it has finished writing, and that the replica is free to garbage-collect older values of x. Write finishes with wok.

As for the replica servers, they have the biggest role in the successful operation of the algorithm (see Figure 12). A replica server responds to client requests for read and write, it secures a value of x, and it garbage-collects old values of x. Also it gossips with all the other replicas about the latest value of x.

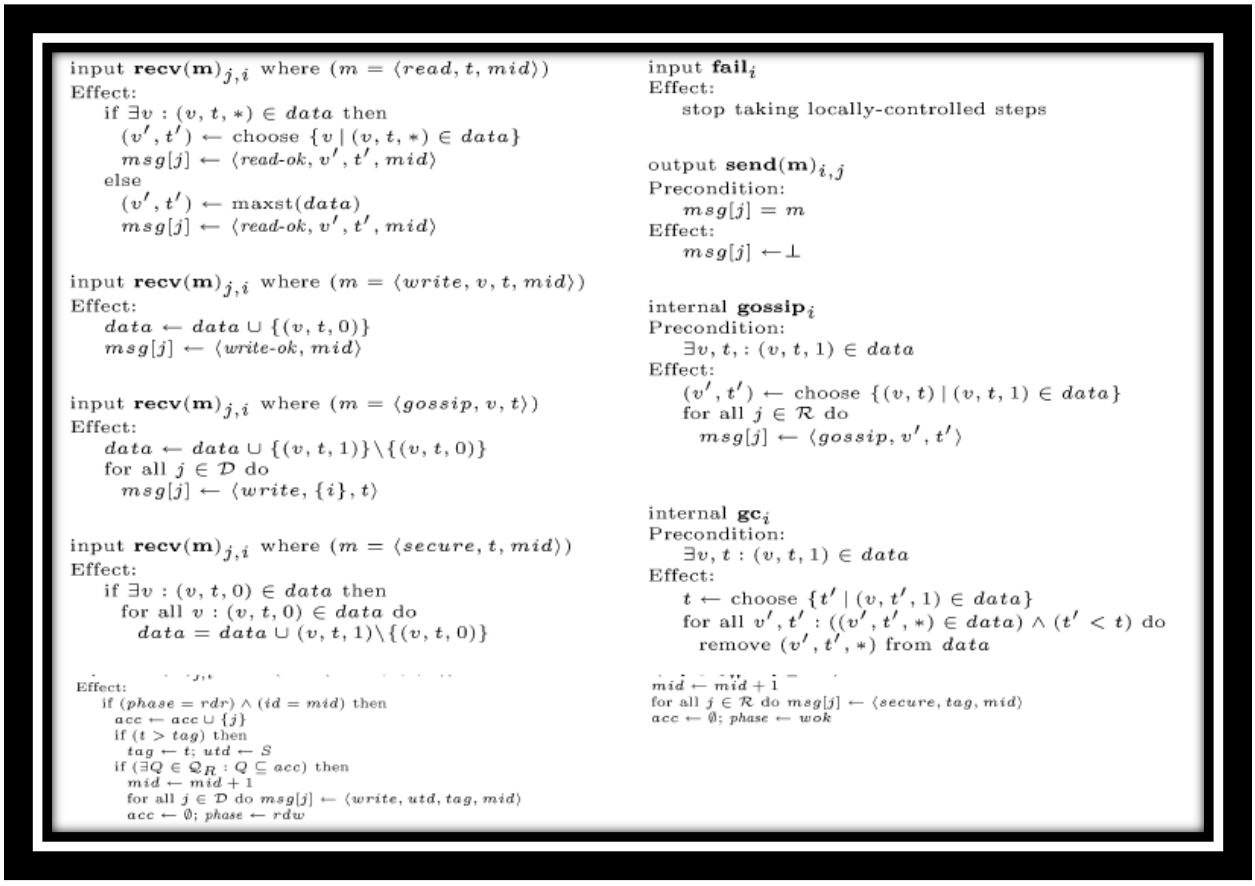


Figure 12. Replicas Transitions [13]

When it receives a message to write a value/tag (v, t), it just add (v, t, 0) to data. If the message is a read message associated with tag t, it checks to find out if it has (v, t, *) in its data. If so it returns (v, t), otherwise it searches its data for the largest secured tag (v', t', 1) and returns (v', t'). If it receives a secure message about tag t, it checks again its data for a triple (*, t, 0) and if there is one, it sets the third argument to 1.

When a replica garbage-collects, it finds a secure value (v, t, 1) in its data, it keeps that, and removes all the triples (v', t', *) with t' < t. On the other hand, when it gossips, it searches its data for a secure value (v, t, 1) and sends (v, t) to all other replica. When it receives a gossip message for (v, t), it adds (v, t, 1) to its data.

Finally the directories' only job is to respond to client requests to read and write utd and tag (see Figure 13). When a directory receives a read message, it returns (utd, tag). When it receives a write message (S, t), it firstly checks if t ≥ tag. If not, that means the request is out of date, and sends an acknowledgment but does not perform the write. If t = tag, it adds S to utd, and if t > tag, it checks whether |S| > f +1, and only then it sets utd equal to S.

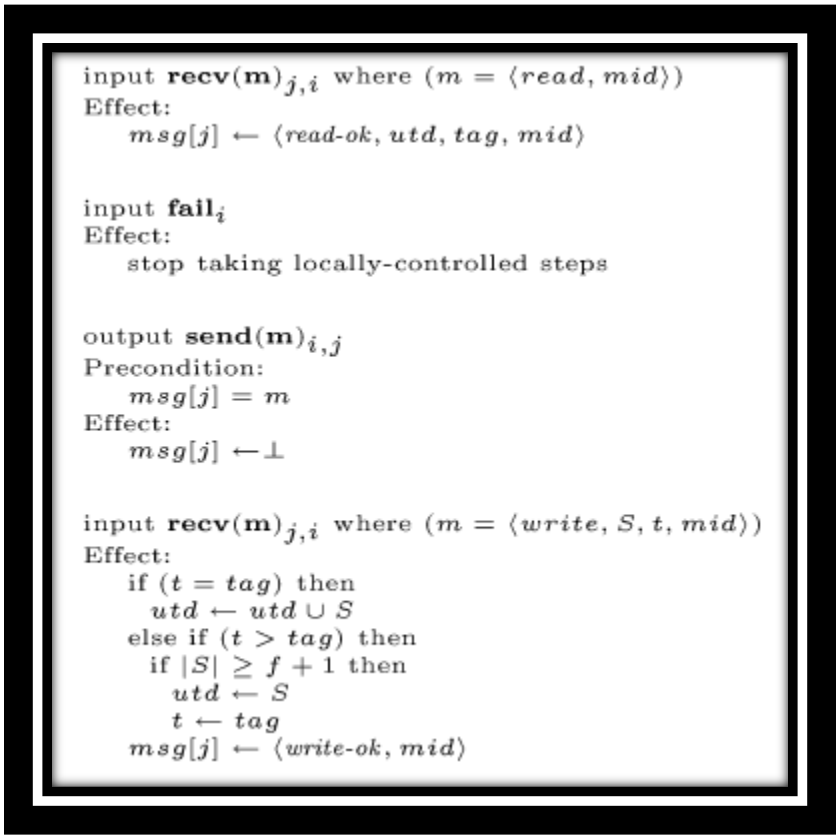


Figure 13. Directories Transitions [13]

3.2 Comparison with Other Implementations

Our implementation has similarities with other well-known distributed file system as well as differences. LDR handles atomicity the same way as the ABD protocol. With two communication phases during reading, one for reading the metadata and the other for writing the newest version back to all the Directory Servers.

Like Coda, LDR allows clients to store files they have read locally on their disk and make changes on them, but unlike Coda, LDR servers do not keep track of which clients have a copy of a file stored in their local space neither it allows servers to send messages to the clients at regular periods of time in order to inform them that they still working on their requests, as Coda does.

Unlike Plan9, LDR has specific roles for its nodes. They can be either clients, or servers (directories or replicas). A client does not have the authority to act as a server for other clients, directories don't count them as file hosts, and they don't include them in the list with the most up-to-date servers that they send back to the requesting clients.

LDR is similar to xFS when it comes to process types. They both separate servers in those that handle metadata and those that deal with the actual data. Similarly, in GFS clients communicate with the master for the metadata information and then with the chunkservers for the operations on the real data. In addition to that the FAB replica-management protocol does something equivalent; it permits actual data reads to be made from any replica, while the other replicas only provide timestamp information.

Like in FAB, LDR follows majority-voting. In more details, when reading or writing from/to directories a client waits for majority of acknowledgements. Also, like FAB, LDR acts in order to protect linearizability. During reading of metadata information, it follows two phases, first it reads and then it

writes the newest version back to the directory servers. On the other hand, unlike it LDR does not offer reconfiguration of the network.

Directory servers in LDR don't need to communicate with replica servers like in GFS, where the master sends Heart Beat messages to them in order to give them instructions and get their stage. Also like GFS, LDR does not store the actual data in its cache due to their size. They cache only the metadata info. Actual data are stored as local files. But in contrast with GFS, LDR does not save the metadata changes in an operation log, like the master in GFS does. Though it is a good plus to have each server restore its status from a log file after a crush, it has not the importance level that it has in GFS due to the fact that the master there is only one and represents a single point of failure, and this is how they deal with it in case of failure. Also in LDR replicas are not divided into primaries and secondaries as it happens with the chunkservers in GFS. They all follow the same behavior when they are called to service a client.

Like Belisarius, LDR has a fix number of servers and an arbitrary number of clients. That's because each server creates a thread for every connection with a client. In contrast with Belisarius, security is a field not covered in a satisfying level in LDR. Except from the digest check on the file, LDR does not offer confidentiality protection like it happens in Belisarius fault tolerant system, which provides protection even from Byzantine nodes. Something the two systems have in common is that they both have the failure factor, but in case of failure the clients in Belisarius wait for $2f+2$ responses, and if not only from $f+2$. On the other hand, LDR clients for $f+1$ responses from replicas regardless if a replica has failed or not.

When it comes to consistency, only LDR and FAB offer the strongest type of consistency, all the other implement weaker types than those two and need to use mechanisms like locking and group communication in order to handle concurrent operations.

3.3 Implementation

We have implemented three types of processes, one client, one directory server and one replica server. We used the C programming language and our implementation was tested on CentOS Linux Operation System (in localhost) and in Fedora OS (in PlanetLab).

3.3.1 Client

Client nodes may be Readers or/and Writers. Every client keeps the IP addresses and communication ports of both directory and replica servers stored in a file. They create reliable TCP communication sessions with them every time they wish to read or write a file. With directory servers, clients exchange control messages, while with replica servers they exchange file content blocks.

A Reader goes through three communication phases (see Figure 14), two with the directories and one with the replicas. During the first communication step, it creates |D| TCP sessions, one for every directory server. It increases the mid (message counter), and sends a <“read_r”,name,type,mid> message to them. Then it waits to receive majority of answers. Majority equals to |D|/2 +1. Every time it receives one, it increases the acc counter, and checks if a possible max tag for the requested file has been found. If it has, it stores the tag and the associated updated set of replica servers. In the case that the requested file does not exists in the Directories' list, they respond with the appropriate message, and the client informs the user for the absence of the file.

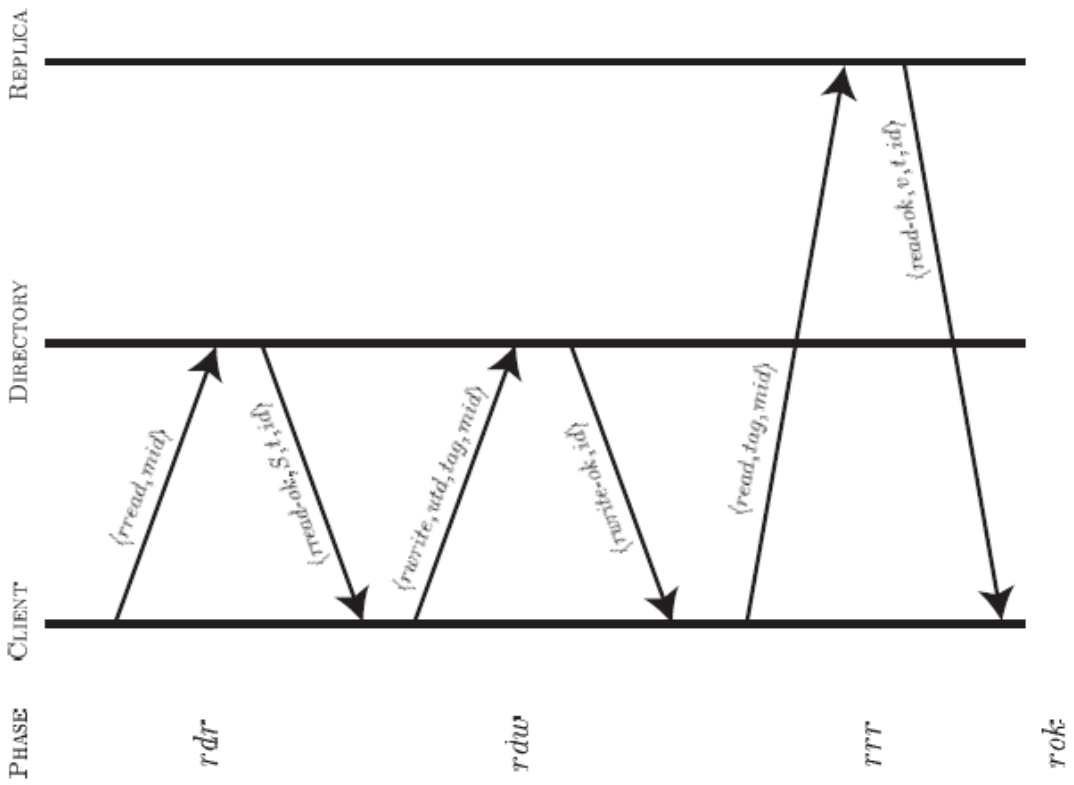


Figure 14. Client Read Operation [13]

When majority of answers are received, the clients closes all socket connections with the directories, and moves on the second communication step, which is to inform all directories about the most up-to-date file's tag and updated set of replicas. It increases the mid counter, and once again creates |D| TCP sessions with all the directories. It then sends them a message <“write”, name, type, S, tag, mid> to each of the directories. S is the set with the most up-to-date replicas. Every time it receives a respond, it increases the acc counter. When it receives majority of acknowledgments, it closes all the socket connections, and moves to the third and final communication phase, which is to read the file's content.

To read the file's content, it must communicate with a random replica. It increases the mid counter, it chooses one replica from the updated set S randomly, and created a TCP session with it. After that, it sends a <“read”, name, type, tag, mid> message to that replica. Replica will send to the client a control

message with the tag of the file, its size, the number of blocks it will send, and the size of the last block. If the control message is correct, the client will store the tag received from the replica, because due to concurrent atomic writes the received tag may be greater than the one the client sent to the replica.

The client must reply back with an acknowledgment message <“write-ok”, mid> and create a file with the name and type it asked earlier. Now is in position to start receiving the blocks of the file. Blocks as we explained before, have a fix size of 944 bytes. When it receives all the blocks, it closes the file, it creates its digest and sends to the replica a <digest, mid> message.

On the other hand, a Writer goes through three and a half communication phases (see Figure 15), two with the directories and one and one half with the replicas. During the first communication step, it creates |D| TCP sessions, one for every directory server. It increases the mid (message counter), and sends a <“read_w”,name,type,mid> message to them. Then it waits to receive majority of answers (majority equals to |D|/2 +1). Every time it receives one, it increases the acc counter, and checks if a possible max tag for the requested file has been found. If it has, it stores it. In the case that the requested file does not exists in the Directories' list, they respond with the appropriate message, and the client knows the file that is about to write is a new one. When it receives majority of answers, it closes all the TPC sessions.

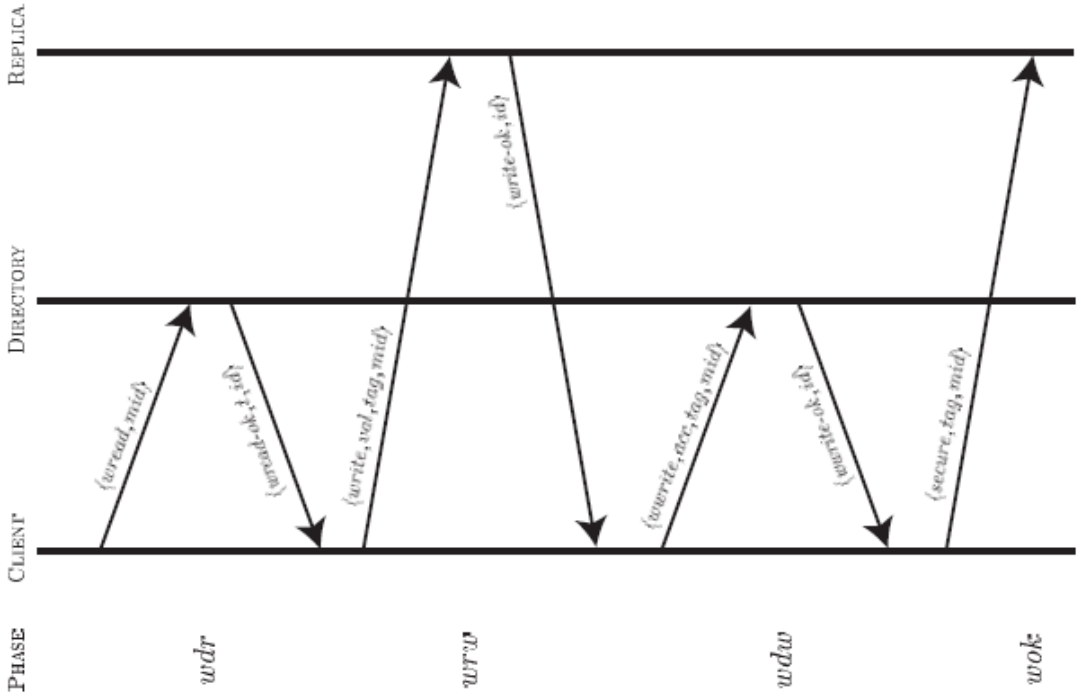


Figure 15. Client Write Operation [13]

The writer then moves to the next communication step, which is to send the file to the replicas and wait for *f+1* acknowledgments; *f* indicates the assumed upper bound on the number of copies that can fail. It increases the mid counter and the version of the file, and then opens the file to count in how many block it must break it. Then it creates |R| TCP sessions one for every replica, and sends them a <"write", name, type, tag, file_size, blocks#, last_block_size> message. It waits to receive acknowledgment from all. When it does, it starts sending the file content broken into blocks of size 944 bytes. After it finishes sending all the blocks it waits to receive one acknowledgment. The acknowledgment will contain the digest that the replica has calculated. Then it will perform a check to see if the digest it will produce is equal with the digest received. If the two digests are the same, it knows that at least one replica has a correct version of the file and closes all the communication sockets with the *f+1* replicas.

On the third communication step, it informs all directories about the most up-to-date file's tag and updated

set of replicas. It increases the mid counter, and creates |D| TCP sessions with all the directories. It then sends them a message <“write”, name, type, S, tag, mid> to each of the directories. S is the set with the most up-to-date replicas. Every time it receives a respond, it increases the acc counter.

When it receives majority of acknowledgments, it forwards to the third and one half communication step. It increases the mid counter, and creates |R| TCP sessions, one with each replica server. It then sends to all a <“secure”, name, type, tag, mid> message, in order to secure the file it just wrote. When it does that, it doesn't wait to receive any responds from the replicas, it just closes all the communication sockets it has opened with them.

3.3.2 Directory Servers

Directory Servers “talk” only with the clients. They have no communication with the Replica servers. Their responsibility is to store the files’ metadata info. They do their calculations in a transparency manner, and when clients ask information for a file they always return the most up-to-date result.

When the directory server begins it service, it follows some initializing instructions read from an initializing file. Then it setups a TCP socket where it listens for file metadata requests from clients. It can accept maximum t concurrent requests, where t is at most the number of clients in the system. Every time it accepts one, it creates a child threat that is responsible for serving a client.

There are two types of requests, that a threat can receive: <action, file name, file type, mid> and <action, file name, file type, S, tag, mid>. The first is a read request. The action field equals to “read_r” in case that the client is a Reader, or “read_w” in case the client is a Writer. The file name field takes the file's name, the file type the given type of the file and mid is a message counter. On the other hand, the second

type of request is received when it comes to a write operation. The action field equals to “write”. The file name, file type and mid fields are similar with the corresponding fields in a read request. S is the set with the most up-to-date replicas and |S|=|R|. S takes values in the scope of {0,1}. S[i]=1 if replica i has the latest version of the file. If not, it becomes 0. Tag is a composite field that includes the version of the file and the IP of the writer.

If the action is a “read_r”, then the directory realizes that it “speaks” with a reader that wants to read the metadata information. It searches its list for the asked file and type. If it finds it, it responds sending a <“read_ok”, S, tag, mid>. If not it sends a <“read_ok”,0,0,mid> to inform the reader that the file it asks for does not exist in any of the replicas.

In case of “read_w” action, the directory includes all the fields it sends for a “read_r” action, except from the S field, because the writer does not need the set of the most up-to-date replicas, it just needs the latest tag. If the directory does not find the asked file, it sends a <“read_ok”,0, mid> to inform the reader that the file it asks for does not exist in any of the replicas.

Finally, if it receives a “write” request (from a reader or a writer), it searches its list, to see if the file node exists. If it does, it checks the tags. If the received tag is the same as the tag stored, it assumed that the file writer hasn't changed, and it unites the S set stored with the S set received. In the case that the two tags differ from each other, if the received tag is greater that the stored one(greater version or equal versions but greater writer IP), then if the number of updated replicas is equal or greater than $f+1$, where f is the number of accepted replicas failures, it sets its S and tag equal to the received S and tag. If the directory does not find the file in its list, it creates a new node giving it the information it has received from the client. When it finishes it responds with a <“write_ok”,mid> back to the client. During the “write” action, each threat attempts to lock the critical area, to prevent two or more threats to modify the directory's list of file nodes.

3.3.3 Replica Servers

Replica servers communicate only with the clients. They are not aware about directory servers' existence. A replica stores files of different types (*.txt, *.doc, *.pdf) and images (*.jpg), and offers transparent read and write atomic operations to its data.

Replicas start their operation having a "hello" file stored, and listens for TCP connections from clients. Every time a replica accepts one, it creates a threat in order to service the client's request, which may be for read, write, or secure.

When it receives a "read" request, it gets a message with <read, name, type, tag, mid> format. Read is the action, name and type give the requested file, tag gives the version and the writer IP address, while mid is the message counter. Then, the replica's thread searches replica's list of stored files to check if the requested file is included there. If it doesn't find the given tag (garbage collect has been preceded), it repeats list searching but this time it seeks for the max secured tag, for the requested file. Either way, it opens the file and counts the number of blocks that this must breaks into.

Each block has a default size of 944 bytes. The size has been chosen after larger sizes have been tested both locally and in Planetlab [15]. Planetlab made things a bit difficult, because with larger than 1 KB sizes and more than one clients, the TCP socket had the bad habit of breaking the message in smaller blocks, something that put the operation of the server in risk of failure, because the replica not been aware of that, it will expect to receive the number of blocks the client declare in its control message, not a greater number, so it won't receive all the data. Also we wanted the file to be divided by 8, to avoid any problems with binary files.

It sends then a control message back <"read-ok",mid, tag, size, blocks#, last block's size> to the client,

the size of which is not greater than 100bits. The two last fields indicate the number of blocks to be send, and the size of the last block, which is less or equal than 944 bytes. It waits to receive an acknowledgment message from the client and then starts sending the file block-by-block. It then waits for another acknowledgment message from the client, that contains the digest that the client has retrieve after passing the file from a hash function. Replica then compares the digest received with the digest it produces using the same function, something like a ckecksum procedure. The hash function is an optimization of a simple hash function from the book by Robert Sedgwicks [35], that we modified it to accept files instead of values. If the two digests mach, the replica knows that it has send the file undamaged. If the file has been corrupted, the replica does not retransmits the file again.

When the writer receives a write request, it first gets the metadata info in a control message, no more than 100bits. Metadata includes the file name, the file type, the tag of the file, its size, the number of blocks it will receive, and the size of the latest block. It then creates a new node on its list, and stores inside all these information. After that, it responds with an acknowledgment message of <“write-ok”,mid> format.

The first thing to do after that is to create a file. To separate it from the other, it merges on the file its tag. It opens it, and while the number of blocks is not equal to the expected, it keeps receiving and storing the blocks of the file. When the receiving process finishes, it close the file, takes its digest and sends the message <digest,mid> back to the client.

In the case it receives a secure request, it does two things. First it seeks the file list to find the requested file, and sets its secure level from 0 to 1. That means, that the replica has received the correct content of the file during write operation, and know is in position to send this file when a reader requests for it. In order to do that, it removes the previous file from its memory, and renames the one with the newest tag to its formal name without the tag extension. If it doesn’t find the file it just removes the file from the local disk.

The second step has to do with the garbage collector. During this step, the replica goes through every node of the list, and removes those nodes that refer to the requested file and have smaller tag from it. Every time the list is being modified, this part of the code is consider as critical area and is lock to prevent two or more threats from modifying it concurrently.

3.4 File Handling

In our implementation, the local disk of each replica contains only one file that is being used for read, and has the actual name of the file. Besides that file, there are multiple other files in the replica that have a temporary name and are in the phase of writing from a specific client. When a client sends a secure message to the replica, the thread that handles the income message, gets in a critical area where only one thread can insert at a time, and removes the old version of the file and replaces it with the one in the secure message, giving it its official name. In case it doesn't find the file in the file list that all threads shared, it understands that some other thread has already removed it as old version. A "write" thread can only remove a file node from the list, not the actual file from the local space. The actual file is being removed later by the thread that has been created to handle the secure message for this file.

3.5 Debugging

For debugging our programs, we first needed to install the Gnome Compiler Collection (gcc) [36] in each node in Planetlab [15] which we used in our scenarios.

To debug a client we used gcc –o executable_name file_name.c.

To debug a replica server and a directory server we used gcc –o executable_name file_name.c –lpthread,

where –lpthread is needed when debugging programs with threads in.

Because of the complexity of the algorithms we faces a lot of cores. For this reason we used the gdb[36] in order to find the faults easier. Gdb steps through source code line-by-line or even instruction by instruction. In additon, it also helps to identify the place and the reason making the program crash.

The order of instructions that we used for gdb are:

i. ulimit -c unlimited

ii. lunch our program and let it crashed

iii. lunch the debugger with gdb <executable_name> <corefile>

iv. run

v. bt

3.6 Encountered Problems

During the evaluation on PlanetLab we faces some problems. The time() function could not give us accurate results. For this reason we used a more accurate on PlanetLab function, the gettimeofday() function that gives current time in seconds. Another problem faced was the bind problem. There were cases that we needed to close a connection abnormally. That caused, the socket to stay bind to the IP address. The next time we run the executable, the socket could not re-bind to that address again. With the use of lsof command we managed to find the open connection and remove it after an abnormal termination.

We also had a problem writing and reading to/from binary files using strlen(). We found out that using strlen() in fwrite or fread was not correct. Instead of that we used sizeof() to get the actual content of a file in blocks. Something also noteworthy, is that we weren't able to compile a source code locally and run the executable on PlanetLab. The two compilers didn't match. So we needed to install the gcc

package on PlanetlLab node, and compile the code up there. Some nodes did not accept package installations on them due to the gpg check. For this reason in some cases we needed to modify the installation command, and do the installation using the --nogpgcheck. Also, sometimes the executable could run locally without any problem caused, but when run it on PlanetLab the code appeared to have segmentation fault, something that the locally run did not show.

Finally, we could not use on PlanetLab block sizes greater than 1 KB. While locally it was okay to do that, on PlanetLab if used more than 1 KB block size, this caused the transferred packages to break into smaller package sizes, that has a result the receiver side to end up with a segmentation fault.

CHAPTER 4

Experimental Evaluation

In this chapter we provide the experimental evaluation details. Some information about Planetlab [15], the scenarios, and the results and conclusions of this thesis.

4.1 Planetlab Platform

Planetlab [15] is a global network or platform, where researchers and academics are able to test large-scale applications for distributed storage, network mapping, peer-to-peer systems, distributed hash tables and query processing. Since the beginning of 2003, more than 1,000 researchers at top academic institutions and industrial research labs have used PlanetLab to develop new technologies for distributed storage, network mapping, peer-to-peer systems, distributed hash tables, and query processing. PlanetLab currently consists of 1163 nodes at 548 sites.

We used the Planetlab EU [15] for testing the performance of our implementation because it provides us with a “real” asynchronous environment where failures may appear and nodes can crash, to see if our implementation is in position to respond to these challenges. As of January 2012, PlanetLab EU consists of 306 nodes at 152 sites.

We have to admit that it has revealed us faults that haven't appeared during our local testing, and has also shown a different and unwanted behavior in scenarios where the size of the files was larger than 1KB.
Nodes on Planetlab are managed via the OPENSSH protocol and can only be accessed using a setup

private/public key pair. We created this key locally on our UNIX account in the University of Cyprus. The command needed for creating this key is the follow:

ssh-keygen -t rsa -f ~/.ssh/id_rsa

Planetlab needs some time to distribute the key to the selected nodes, so if one creates this key, it must be uploaded it in his/her planetlab account, and right after you do those two steps you try to connect with a node, a failure message will most possible appear to you.

If one wants to add nodes (computers), must go to his/her slice, where there is a list with nodes and choose the ones he wants. Of course, one will notice that in the list will appear and crashed nodes. He/She will choose the nodes that their status shows that are boot. Because we are using the EU planetlab, one is preferred to choose the ones that are in PLE authority.

To login to a node with SSH, one must give his/her slice name as the login name (e.g. princeton_test1), the path to his/her private key file (e.g. ~/.ssh/id_rsa), and the node to login to (e.g. planetlab-1.cs.princeton.edu):

ssh -l princeton_test1 -i ~/.ssh/id_rsa planetlab-1.cs.princeton.edu

To upload his/her code to a planetlab node, one will have to do it locally using the following scp command:

scp -i ~/.ssh/id_rsa -r test1 princeton_test1@planetlab-1.cs.princeton.edu:

In case someone wants to modify his code while being on a node on planetlab, he can do this if he has

first installed for example the emacs editor on that node.

4.2 Scenarios

We have run five types of scenarios, each of which gave an extra hand for the conclusion we took for the implementation. For every scenario we took the average read/write operation latency. The operation latency measures as the time needed for an operation to complete (the time from tis innovation until its completion). For every scenario we run twenty (20) operations, from which we recycled the min and max time results and took the average time of the rest. We run each scenario three (3) times and took the average of the average time results that each of it gave, to create the graphs. During all the scenarios we used a fixed message size of 944 bytes, and small control messages. For this reason we didn't check how the size of the exchanged messages affects the results. In addition to that, every process (reader, writer, replica server, directory server) had a fixed role in the scenarios, and we set each one to run on a different physical PlanetLab machine. Also we checked the average ping delay and we obtained that it is 35ms.

Scenario 1 – Readers

In the Readers Scenario, we took the average read operation latency for 10, 20, 30 and 40 Readers. The number of Replicas and Directories, and the size of the file were fixed for all four choices. Three Replicas, three Directories and a .doc file of 1KB size.

The purpose of this scenario was to see if and how the read operation latency changes when increasing the number of concurrent readers. How this affects performance and consistency of the system.

Scenario 2 – Writers

In the Writers scenario, we used 5, 10, 15 and 20 writers and no readers. The number of Replicas and Directories, and the size of the file were fixed for all four choices. Three Replicas, three Directories and a .doc file of 1KB size.

The purpose of this scenario was to see if and how the write operation latency changes when increasing the number of concurrent writers. How this affects performance and consistency of the system.

Scenario 3 – Replicas

In the Replicas Scenario, we used 10 writers working in parallel with 20 readers for a fixed file size of 1KB and a fixed number of 3 Directory Servers. And we took the average read and write operation latency for 3, 6 and 9 Replicas, having the Failure factor *f* increased by one every three Replicas.

The purpose of this scenario was to see for a small number of clients (thirty) what number of Replicas gives the best performance and whether by increasing the Replicas and Failure factor *f,* the consistency is affected.

Scenario 4 - Directories

In the Directories Scenario, we used 10 writers working in parallel with 20 readers for a fixed file size of 1KB and a fixed number of 3 Replica Servers. And we took the average read and write operation latency for 3, 5 and 7 Directories.

The purpose of this scenario was to see for a small number of clients (thirty) what number of Directories

gives the best performance and whether by increasing the Directories' number the consistency is affected.

Scenario 5 – File Size

In this scenario, we tested the performance of our implementation using 20 Readers, 10 Writers, 3 Replicas, 3 Directories, but different file sizes. We took the average read/write operation latency, for 1KB, 100KB and a 250KB file.

The purpose of this scenario was to see the strengths of our implementation. The performance it has with large data objects and a block size of 944 bytes. And using this block size until what file sizes the implementation works correct when having concurrent writers and readers.

Scenario 6 – Sequential vs. Concurrent Readers

In this scenario, we tested the performance of our implementation using 10 Readers, 3 Replicas, 3 Directories, and a doc file of 250KB.

The purpose of this scenario was to see how concurrency affects the performance of the algorithm during a read operation. We took the average read operation latency using 10 sequential readers and 10 concurrent readers.

Scenario 7 – Sequential vs. Concurrent Writers

In this scenario, we tested the performance of our implementation using 10 Writers, 3 Replicas, 3 Directories, and a doc file of 250KB.

The purpose of this scenario was to see how concurrency affects the performance of the algorithm during a write operation. We took the average write operation latency using 10 sequential writers and 10 concurrent writers.

4.3 Results

In this section we discuss the results obtained after we run our implementation for each of the scenarios described above.

Readers Scenario

Figure 16, shows in the x-axis the average operation latency in secs and in y-axis the number of readers run. Ten concurrent readers finished in 0.58 secs, twenty in 0.6 secs, thirty in 0.61 secs, and forty in 0.63secs.

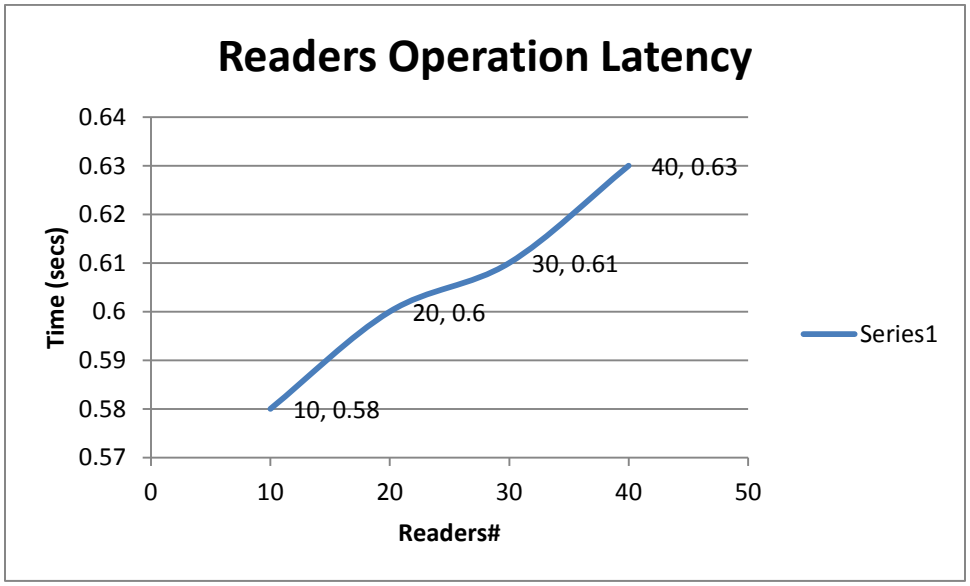


Figure 16. Readers Operation Latency.

What we observe in this scenario is that the time needed for reading is independent from the number of concurrent readers, but it depends on the size of the file and the block size in which this files breaks into during the transfer. We can have as many concurrent readers as we wish, because the implementation creates a thread for each of them and they all execute their operations the same time without any locks. On the other hand, we can see that even though we have run this scenario on the Planetlab evaluation platform, this result is subjective if when increasing the size of the file we leave fixed as it is the block size. We can see it as an objective result only if we increase these two variables in a balanced manner. When it comes to the obtained results, 0.6secs are needed for 20 concurrent readers to read a file of 1KB size. Given that for a ping we need an average time of 36.7ms and a read operation takes three round trips, then when we take an average operation latency of 0.037secs, that means that we need an average time of 0.11secs for the three communication phases to complete. 0.49secs average processing time is reasonable. We believe that the implementation gives satisfying average read operation latency for twenty concurrent readers to complete their operation.

Writers Scenarios

Figure 17, shows in the x-axis the average operation latency in secs and in y-axis the number of writers run. Five concurrent writers finished in 0.7 secs, ten in 0.81 secs, fifteen in 0.86 secs, and twenty in 0.81secs.

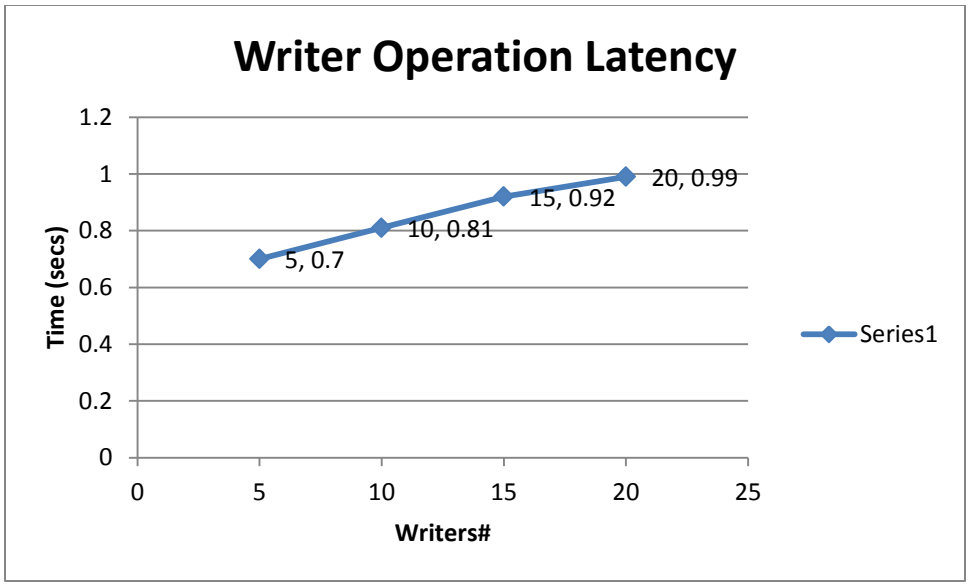


Figure 17. Writers Operation Latency.

What we observed in this scenario is that the time needed for writing depends on the number of concurrent writers, and also on the size of the file and the block size in which this files breaks into during the transfer. The number of concurrent writers affects in a small degree the taken results, because we had to lock a critical area where file deletion take place. On the other hand, this result subjective if when increasing the size of the file we leave fixed as it is the block size. We can see it as an objective result only if we increase these two variables in a balanced manner. When it comes to the obtained results, given that for a ping we need an average time of 36.7 ms and a write operation takes three and a half round trips, then when we take an average operation latency of 0.81secs, that means that we need an average time of 0.13secs for the three communication phases to complete. 0.68secs average processing time is reasonable for ten concurrent writers. We believe that the implementation gives satisfying average write operation latency for ten concurrent writers to complete their operation.

Replicas Scenario

Figure 18, shows in the x-axis the average operation latency in secs and in y-axis the number of replicas run. The writers graph shows the average write operation latency while the readers graph the average read operation latency. For these scenarios we chose a fixed number of ten concurrent writers and twenty concurrent readers, and we tested the implementation for 3, 6, and 9 replicas.

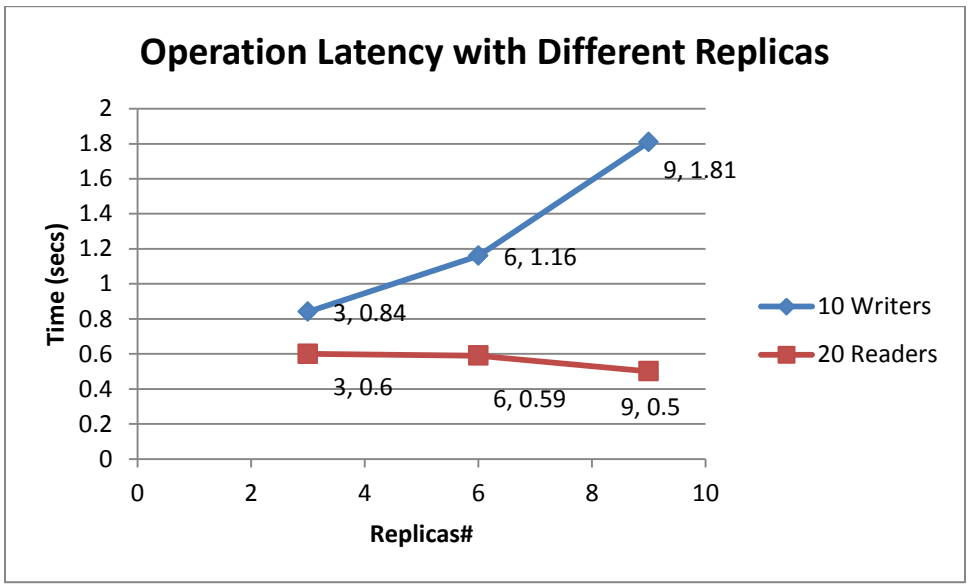


Figure 18. Operation Latency with Different Replicas.

We observe that by increasing the number of Replicas and the *f* factor, we get slightly smaller times during the readers' operation and higher times during the writers' operation. This happens because by increasing the R and *f*, writers must write the file's content to doubled Replica servers and wait plus one responses, while readers have more options now on choosing a random replica server. We decided that the reduction of time needed for reading is not worthy of putting extra time on writing. So we continued with Replica# = 3. We believe that this result is objective when it comes to the increase of the operation time needed and subjective on the values that we take. Also when increasing the number of replicas we most affect negatively the write operation latency. And this is because despite the number of Replicas, during

read we only read the file from one replica, but when it comes to writing, the numbers of replicas to which we send the file each time is doubled and the number of received responses is increased by one due to the *f* factor.

Directories Scenario

Figure 19, shows in the x-axis the average operation latency in secs and in y-axis the numbers of directories run. The writer graph shows the average write operation latency while the reader the average read operation latency. For these scenarios we chose a fixed number of ten concurrent writers and twenty concurrent readers, and we tested the implementation for 3, 5 and 7 Directories.

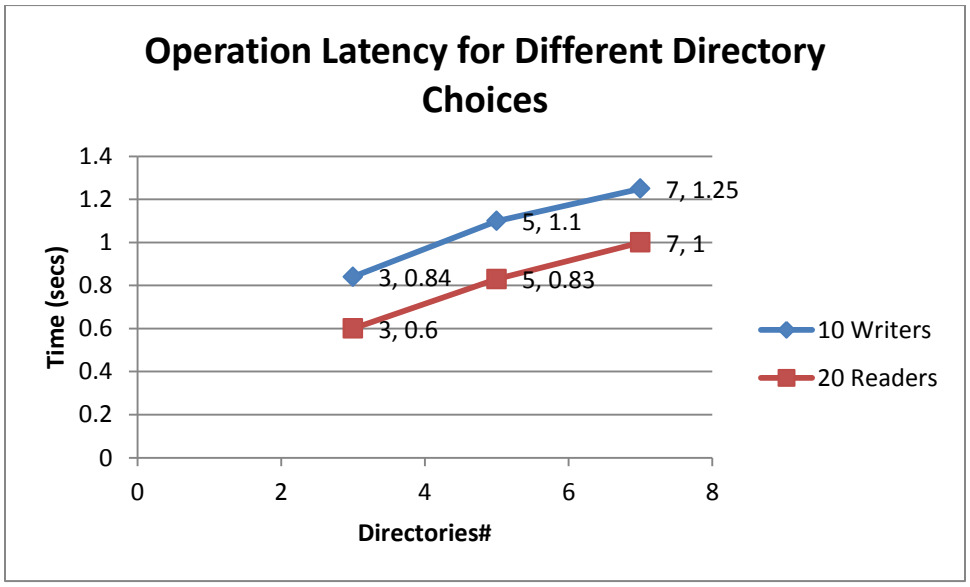


Figure 19. Operation Latency for Different Dirrectory Choices.

By increasing the number of Directories to 5 or 7, we don't succeed on getting betters results. On the contrary it takes more time for readers or writers to complete their operation. For this reason we decided to continue having the number of Directories equal to 3. The results we have taken are objective for these numbers of Directories, because the size of the exchanged control messages is fixed and the operation

latency does not have anything to do with the type of the operation, but only with the number of Directories because a client must communicate with all of them. Also for a small number of clients, three Directories is a satisfying number. The time difference between the two types of operation is due to the fact that a writer has to communicate with the replica servers one extra time during the operation to send them a secure message, something that adds 0.065secs on time (the average ping is 36.7ms). Writers also have a critical area, causing them to wait for their turn to insert.

File-Size Scenario

Figure 20, shows in the x-axis the average operation latency in secs and in y-axis the sizes of the files used in KBs. The writers graph shows the average write operation latency while the readers the average read operation latency. For these scenarios we chose a fixed number of ten concurrent writers, twenty concurrent readers, three replicas and three directories, and we tested the implementation for three different document files of 1KB, 100KB and 250KB size.

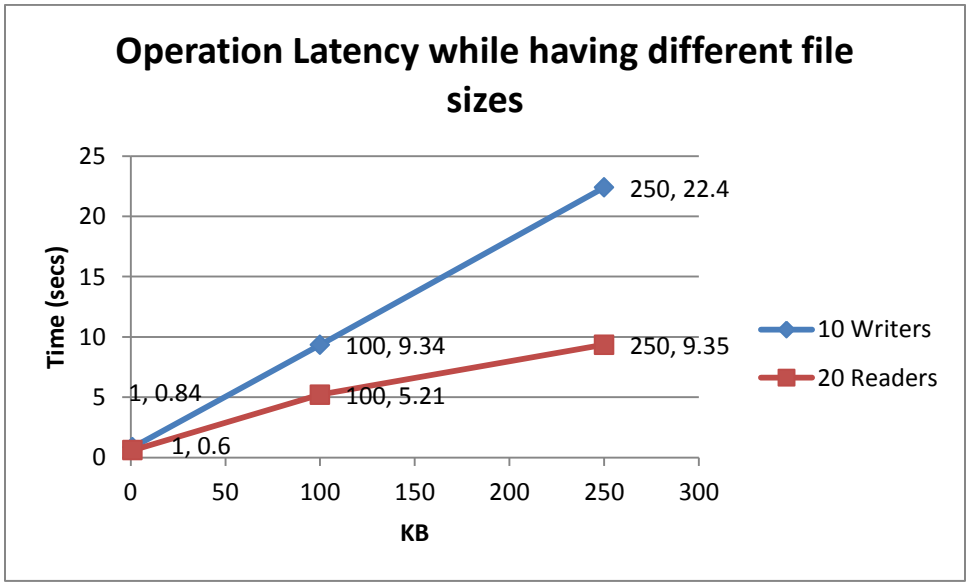


Figure 20. Operation Latency while having different file sizes.

Using a message block size of 944 bytes, these are the results the system gave for 1KB, 100KB and 250KB file. If we could use a greater block size, we would probably get smaller times. As we said above to get objective results one need to keep a balance on block size and file size variables. It is not efficient at all to break a file of 250KB size into 265 blocks, and it is not a surprise that we weren't able to test greater file sizes due to connection timeouts on clients.

Sequential vs. Concurrent Readers Scenario

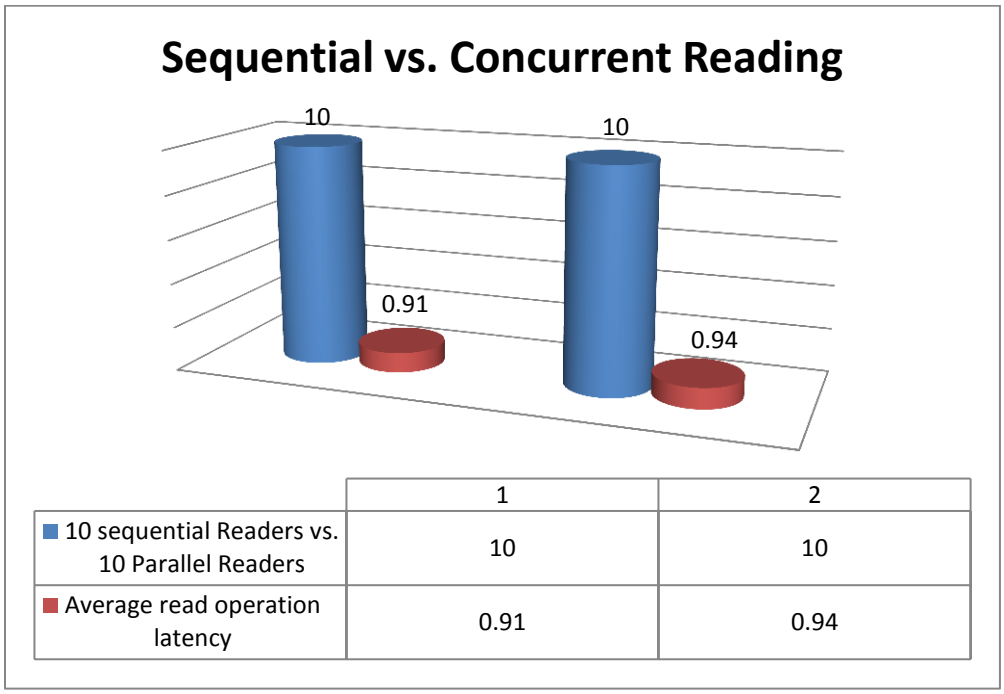


Figure 21: Average read operation latency using sequential and concurrent readers

In this scenario we obtained the results first using 10 sequential readers and then 10 parallel readers. Both have run 20 read operations on a document file of 250KB size. As we observed from the results, 10 concurrent readers need almost the same amount of time as one sequential reader. That means that by doing concurrent reading we put a plus on the implementation's performance, because instead of 10x0.91=9.1secs we need only 0.94secs.

Sequential vs. Concurrent Writers Scenario

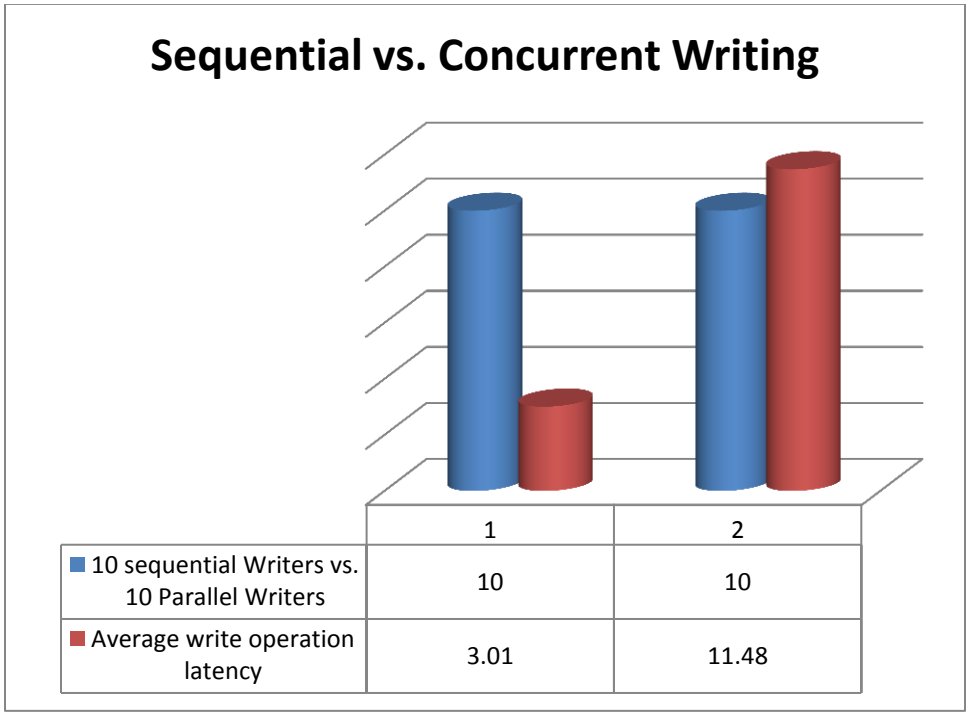


Figure 22: Average write operation latency using sequential and concurrent writers

In this scenario we obtained the results first using 10 sequential writers and then 10 parallel writers. Both have run 20 read operations on a document file of 250KB size. As we observed from the results, by doing concurrent writing we put a plus on the implementation's performance, because instead of 10x3.01=30.1secs we need only 11.48secs.

4.4 Evaluation Conclusion

The performance of the implementation, as expected, was affected by our implementation choices and the particularities of the Planetlab evaluation platform.

When it comes to the implementation we weren't able to avoid the critical area locking during securing of the file and the garbage collection. This had as a result the rest of the writers to wait each time one was doing that. We did that in an attempt to prevent a thread from erasing a file from the local disk while another was reading its content. We didn't find a way to avoid this.

Also while locally we could run the implementation with greater than 1KB block sizes, on Planetlab it seems that at least using TCP the size of the sending block is limited, something that has affect the performance in the greater degree. This is most probably due to a limitation putted by the PlanetLab developers that one needs to investigate.

In order to be able to compete with the performance of the distributed file systems mentioned in chapter 2, for example GFS (that has a block size of 8GB), we need at least a block size of 1MB. We tried to do that, but the TCP protocol we used for data transfer on Planetlab kept breaking blocks greater than 1KB into smaller packages, something that caused segmentation faults and even if it didn't, this kind of behavior is not wanted when trying to achieve greater block size transfer. Maybe a connectionless protocol like UDP would make things easier.

Despite the above, in all cases the algorithm never violated atomicity and its performance was reasonable, given that PlanetLab's cost of communication time (the average ping is 36.7ms). A final conclusion that can be made from our evaluation is that the algorithm can be implemented and efficiently run on an adverse, real planetary-scale network such as PlanetLab. Hence, at some extend, our work demonstrates the practicality and potential of LDR.

CHAPTER 5

Conclusion and Future Work

In this final chapter we present our conclusions and we give recommendations for a future work on the implementation.

5.1 Summary

We have managed to implement algorithm LDR algorithm and perform an experimental evaluation. The algorithm LDR, achieves (theoretically speaking) the three main goals that any Distributed File System wish to achieve: replication, performance and atomic consistency.

The implementation provides large data objects replication and mechanisms that ensure that this won't affect the linear consistency of the algorithm. Also it does that in a transparent way to the clients. The key in the algorithm's performance is that, we separate the metadata from the real data so that we can do cheaper operation on the first ones.

Our implementation has similarities with other well-known distributed file system as well as differences. LDR has a fix number of servers and an arbitrary number of clients. That's because each server creates a thread for every connection with a client. It also has specific roles for its nodes. They can be either clients, or servers (directories or replicas). A client does not have the authority to act as a server for other clients. Clients do not store the actual data in its cache due to their size. They cache only the metadata info. Actual data are stored as local files.

From the evaluation we may conclude that our implementation does provide transparent file replication, and strong consistency guaranties, and that LDR in the tested cases is efficient.

5.2 Future Work

The system should give better results with higher performance for a larger file block size. The current block size does not allow the system to show its real powers, due to the fact that when trying to use larger file sizes, for example 1MB, replicas aren't able to satisfy all the concurrent clients when using a block size of 944 bytes. With a block size of 1MB for instance, it has the perspective to achieve nugatory operation latency.

Also an interface is needed. For evaluation reasons everything is automatically chosen: the operation, the file to be processed, etc. An interface for user interaction will give better visual results and when this is done we can easily implement an authentication test for a client's rights on a file. Also in case the checksum on a file's digest fails, this can be represented there in better visual way.

During a read operation, a timeout is needed in case the read operation from one replica fails, so it can choose another. Also something we haven't touched is that our servers are stateless. In case of fail they won't be able to reboot on their previous status. A log file would be very useful if we want to face this problem. Finally something we haven't implemented from LDR algorithm is replicas gossip function. We didn't need it in our implementation due to the fact that nodes on Planetlab don't crash regularly, but in a distributed file system with large numbers of clients, it may be necessary.

In addition to that, a replica server or a writer in the current implantation they both send first a message with the metadata info and then the blocks with the actual content of the file. In a newer version of the

implementation, metadata control message may be joined with the first real content data message, for performance purposes, especially in the case where writer sends the metadata info and before it starts writing to replica servers it first waits acknowledgments from all, an act not wanted due to the failure possibility on nodes. Also, though the designer of LDR they clearly say that a reliable data trasnsfer protocol must be used, one could create a version of LDR where both TCP and UDP transfer protocols are used depending on the type of the transferred object. For example if one wants to add the live streaming option on a stored file, UDP is the most ideal option.

REFERENCES

1. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. Journal of the ACM, 42(1):124-142, January 1995.
2. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3): 463-492, July 1990.
3. Marcos K. Aguilera, Dahlia Malkhi, Idit Keidar and Alexander Shraer. Dynamic Atomic Storage without Consensus. ACM 978-1-60558-396-9/09/08, August 2009.
4. N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In 5thInternanational Symposium on Distributed Computing (DISC), 2002.
5. L. Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133-169, May 1998.
6. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv., 22(4): 299-319, 1990.
7. Alexander Shraer, Dahlia Malkhi, Jean-Philippe Martin and Idit Keidar. Data Centric Reconfiguration with Network-Attached Disks. LADIS 2010.
8. John H. Howard. An Overview of the Andrew File System. CMU-ITC-88-062.
9. Mahadev Satyanarayanan and Ellen H. Siegel. Concurrent communication in a Large Distributed Environment. ACM 00 18-9340/90/0300-0328, 1990 IEEE.
10. C. Chalupa and J. Coomer. Product Manual: Fibre Channel Interface. ACM 100293070, March 2006, Seagate.
11. Julian Satran, Kalman Meth, Constantine Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. RFC3720: Internet small computer systems interface (iSCSI). http://www.faqs.org/rfcs/rfc3720.html, 2004.
12. Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. On Database Sys. (TODS), 4(2): 180-209, June 1979.

13. Rui Fan. Efficient Replication of Large Data Objects. Thesis 2003.
14. http://www.partow.net/programming/hashfunctions/#RSHashFunction
15. PlanetLab Europe Technical Overview. https://www.planet-lab.eu/files/PlanetLab__Tech_Overview.pdf.
16. P.A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc., 1987.
17. N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97), pages 272-281, Seattle, Washington, USA, June 1997. IEEE.
18. Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication, 1994.
19. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In Proceeding of the 1996 ACM SIGMOD international conference on Management of data, pages 173-182. ACM Press, 1996.
20. Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh and Bob Lyon. Design and Implementation of the Sun Network Filesystem. CA. 94110 (415) 960-7293.
21. D. Walsh, B. Lyon, G. Sager, J. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun network file system. In Proceedings of the 1985 Winter Usenix Technical Conference, January 1985.
22. M. Satyanarayanan, J. J. Kistler, P. Kumar,M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on Computers, 39(4), April 1990.
23. Dave Presotto, Rob Pike, Ken Thompson and Howard Trickey. Plan 9, A Distributed System. AT&T Bell Laboratories, Murray Hill, New Jersey 07974.
24. T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. ACM Transactions on Computer Systems, 14(1), February 1996.

25. Christoph Hellwig. XFS: the big storage file system for Linux. October 2008.
26. Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. ACM 0360-0300 /94 /0600-0145. 1994.
27. Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence and Alistair Veitch. FAB: enterprise storage systems on a shoestring. Hewlett-Packard Laboratories.
28. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. SOSP'03, October 19-22, 2003. Google.
29. Ricardo Padilha and Fernando Pedone. Belisarius: BFT storage with confidentiality. ACM 978-0-7695-4489-2/11. 2011. IEEE.
30. Dan Bogdanov. Foundations and properties of Shamir's secret sharing scheme Research Seminar in Cryptography. University of Tartu, Institute of Computer Science May 1st, 2007.
31. Creating a TCP Stream Socket Application. http://msdn.microsoft.com/en-us/library/aa454002.aspx.
32. Creating a UDP Datagram Socket Application (Windows CE 5.0). http://msdn.microsoft.com/en-us/library/ms881658.aspx.
33. Understanding TCP and UDP. http://www.yaldex.com/tcp_ip/0672325659_ch06lev1sec3.html.
34. UDP: User Datagram Protocol. http://www.pcvr.nl/tcpip/udp_user.htm.
35. Robert Sedgewick. 1946 - Algorthms. QA76.6.S435 1983 519.4 82-11672. ISBN O-201 -06672-6. August 1983.
36. Quick start with C, gcc, and gdb. http://www.cs.cornell.edu/courses/cs2022/2010sp/01-quick.pdf.