TALLINN UNIVERSITY OF TECHNOLOGY
Department of Computer Science
Network Software and Intelligent Systems

Distributed and Replicated File System for Large Scale Data Storage

Graduation paper

Student: Sven Petai
Student Code: 001740LDWL
Supervisor: prof. Tanel Tammet

Tallinn
2007

Author declaration

I declare that this thesis is based on my own work and has not been submitted for any degree or examination by anyone else.

Date: Signature:

Annotatsioon

Tänapäevaste multimeediarohkete veebirakenduste tarvis on vajalik suurte andmemahtude hoidmine viisil, mis oleks kiire, odav, veatolerantne ja omaks ka suurt skaleeruvuse potentsiaali. Sarnane vajadus tekkis ka Elionil peamiselt erinevate hot.ee portaali teenuste sisu hoidmiseks. Rakenduste seisukohalt oleks parim, kui selline andmesalvestuse meetod paistaks tavalise lokaalse failisüsteemina, kuna sellisel juhul ei peaks rakendusi ringi tegema. Seetõttu otsustatigi antud lahendus luua hajusfailisüsteemina, mis kujutab endast paljudest serveritest koosnevad klastrit, kus klient näeb igaühes neist kõiki klastris eksisteerivaid faile justkui lokaalseid ja võib neid ka sellisena avada, lugeda ja muuta. Failisüsteem kannab ise sisemiselt hoolt selle eest, et fail saaks klientprogrammi jaoks õigest serverist kohale toimetatud, kui teda lokaalses masinas ei eksisteeri. Ühtlasi kannab klaster sisemiselt ka hoolt, et iga fail eksisteeriks alati rohkemas kui ühes masinas, mis vähendab andmete kaotamise riski mõne serveri katkimineku korral.

Käesoleva diplomitöö põhitulemus on Elioni hajusa failisüsteemi disaini ja teostuse loomine. Töös kirjeldatakse täpseid Elioni poolseid ootusi sellisele failisüsteemile ja neist tulenevaid disaini otsuseid, samuti analüüsitakse erinevaid olemasolevaid hajusaid failisüsteeme, mille mitmesuguseid jooni kasutatakse ära ka loodava failisüsteemi disainis.

Table of Contents

1 Introduction

This thesis describes the design and implementation of a distributed file system, created for Elion Enterprises Ltd. The file system is intended to be used as a platform for storage intensive web services. At the moment of writing the implementation is finished and has entered an early testing phase.

For most modern web services like galleries, blogs, video sharing and community portals a lot of scalable, reliable and fast storage is needed. In general there are two obvious ways to approach this problem - using expensive proprietary storage solutions or building your own services so that they would know how to handle storage distributed over a large number of servers.

The first solution is not really desirable because of the closed architecture of these proprietary storage solutions which usually translates to vendor lock-in and high costs. The second solution would be completely under our control and cheap, but we would have to reinvent the storage management logic in every application.

What we were after was a middle ground between the two - a file system that looks to services like a normal local file system, but in reality is distributed over many backend servers.

To achieve this we have decided to design our own distributed cluster file system called RPFS. In the following sections we will first discuss what exactly is a distributed cluster file system and explore previous work done in this area. After gaining sufficient background perspective we will turn to explaining the design and implementation of our RPFS.

2 Goals

The file system that we have designed was going to be used mainly for serving user content to the internet - pictures from the galleries, movies, homepages, private files etc. We wanted to be able to integrate it into our current *hot.ee* portal and possibly use it for other services in the future. It was also clear from the beginning that we have to provide at least FTP access to all the content in addition to the portal.

We formulated the following goals for the project:

- **Scalability**: The architecture must be able to scale well both space and network speed wise to loads that would allow servicing millions of users and provide at least several hundred terabytes of space.
- **Fault tolerance**: The system must survive the loss of at least a single node without data loss.
- **Transparency**: We want to have as little special storage management code in our applications as possible, preferably none at all.
- **Low price**: Our solution has to be a lot cheaper than the current proprietary storage solutions in use, to make services that we plan to run on top of it economically feasible.
- **Network usage optimization**: The architecture should be able to take advantage of the fact that we have servers in several cities over Estonia.
- **Manageability**: The storage system should not need any daily manual administration so that replication decisions, load balancing, fault detection and error recovery should be as automatic as possible.
- **Security**: Since our portals have complicated privilege management schemes it is essential to have support for access control lists (ACL).

In addition to the hard goals that have to be satisfied the following soft goals were defined too:

- Currently managing the content and its access privileges is done from the existing portal software and it would be desired to retain the possibility of managing and reading file and directory privileges through SQL.
- The content is going to be served over HTTP and FTP protocols, in the future additional methods might be added. So if at all possible the file system should not make any assumptions about what applications it has to serve.
- There is a very strong possibility that many users will upload the same video or music files. So it would be useful to store only one physical copy of the file and allow multiple virtual copies to reference it.

To give a preliminary understanding of our vision we present the layout (Illustration 1) of the final solution that will be more closely described in section 7.3.

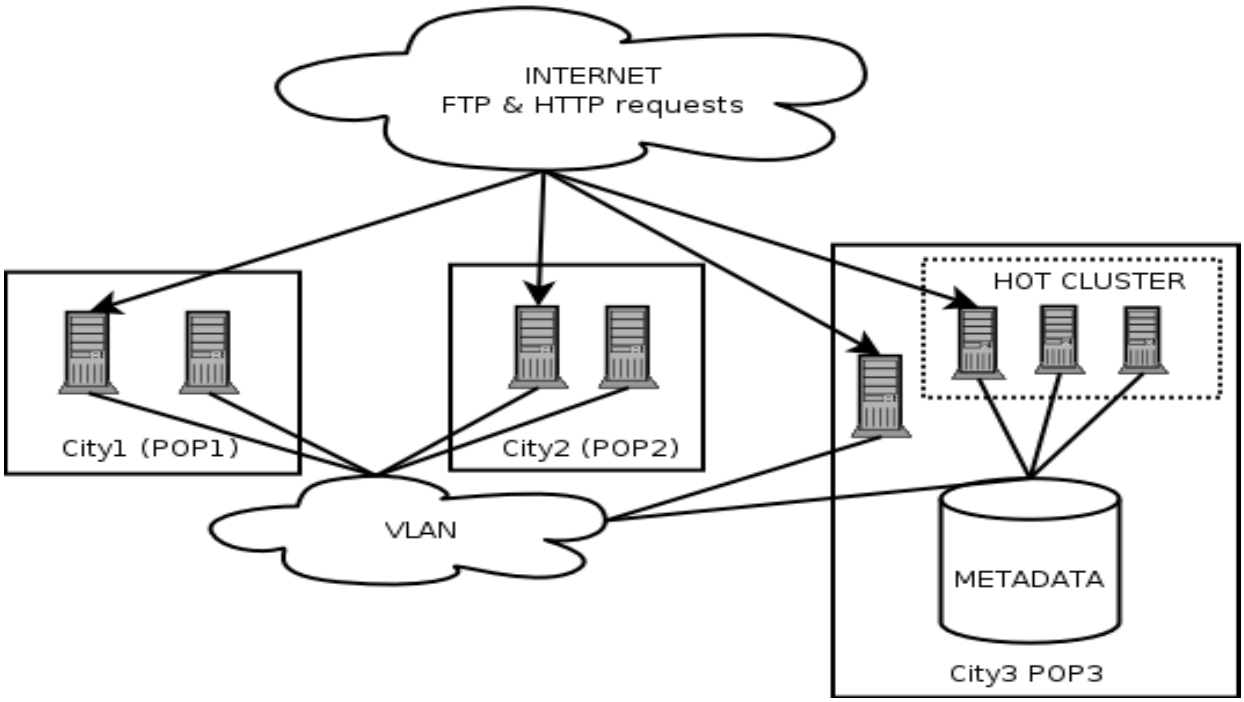

Illustration 1: the overall view of the distributed file system deployment in different cities. The hot cluster is used to manage file privileges and sharing directly in the metadata server. Web pages served by the HOT cluster reference content that is stored in the file system servers located in different cities all over Estonia.

3 Distributed File Systems

This chapter discusses the basic concepts crucial to understanding what file systems in general and distributed file systems in specific are, and what are the main problems for implementation. In this thesis we will concentrate on Unix architecture: unless said otherwise, our descriptions are applicable to operating systems that have Unix roots like FreeBSD, Linux and Solaris.

3.1 What is a file system?

Most programs that we use produce some data that has to be stored on a nonvolatile media (meaning that it will retain data when not powered). This requirement arises partly from the fact that the main memory in our normal computers is volatile and much smaller than the nonvolatile storage media. In order to make such a scheme usable, programs have to write data not currently in use to slower nonvolatile medium for long term storage. Deeply embedded systems with serious resource constraints running only a single application usually implement reading and writing operations directly in the application code. In larger systems it is obviously a good idea to do these operations in a central layer that can be used by various applications only through the programming interface (API). This avoids code duplication, allows us to enforce file access privilege checking, makes it possible to build tools to centrally manage this datastore (set access privileges, quotas etc.) and makes it possible to serialize data changes to provide determinism.

This central layer is called a file system (FS) and is usually a part of the operating system (OS) kernel. File systems operate in logical units called files which is a named collection of related information stored on a secondary storage [1].

Operating systems usually consist of two parts. *User space* or *userland* is the part where applications execute in their own address space, protected from each other. The central part of the operating system is the kernel which is

usually implemented as a single monolithic program running with the highest possible privileges. It is the task of the kernel to check access privileges and provide convenient resource abstractions to the *userland*. Programs running in *userland* request services from the kernel mainly by issuing *system calls (syscalls)* which are basically just functions that the kernel has exported. Usually programs will not issue system calls directly, but will rather call functions from the system library (called *libc* in Unix systems) that in turn may issue system calls to fulfill their function. File system related *syscalls* are handled inside the kernel by the abstraction layer called Virtual Filesystem Interface (VFS) which can map requests to an actual file system component [2]. The real file system component might be local (*ext3, reiser, UFS2*) or remote (*NFS, Coda*).

Usually when an application sends a file to the network it does so by reading a chunk of it from the file system and then writing the read data to the socket. This means that the kernel has to first copy the data to userland and then right back to send it out. This constant data copying can introduce significant overhead so several operating systems have introduced a special *syscall* called *sendfile()* which sends whole files out directly from inside the kernel.

In most file systems files are just byte streams that do not have any structure from the perspective of the file system and the interpretation of the contents is solely up to the program reading it. There are also file systems that assign an internal structure to the file objects – for example in *NTFS* a file is a structured object consisting of typed attributes which can all be accessed independently [3]. The actual file contents are just stored in a unnamed data attribute without imposing structure on its contents. Still other file systems have gone even further and support record orientated files that provide an interface resembling the functionality of RDBMS [4].

In general the primary task of the file system is to abstract away the physical location of the file from the caller so that users do not have to have any idea how the data is stored physically on the media or even where - it might be stored on a local disk or just as well on a machine located at the other side of the world.

Internally most file systems keep information about the actual file contents and associated information separately. File contents are stored just as the application sent it, without paying any attention to its content or adding anything to it. Usually file content is stored in equal sized blocks that are referenced from the file's metadata object. Metadata is the part that stores filenames, access privileges, special flags, various timestamps, location in the directory tree, the actual physical location of blocks on disk and possibly various metainfo like extended attributes and the file type if supported [1].

Besides files there can be many other object types in the file system. The other most common object type is a directory, which allows you to store files or other directories making it possible to create tree like hierachies. File systems that support directories are called *hierarchical file systems* and the ones that store all the files at the same level are called *flat file systems*. Another common file system object is a hard link that is used to reference the same file from multiple directories.

Traditionally an in-kernel metadata object that describes a single file system object is called *"inode"* which is short for *index node*.

The usual I/O flow in a traditional Unix system has the following steps (Illustration 2):

- A userland application calls one of the file system operation functions in the standard system library (libc).
- *Libc* issues a syscall
- File system syscalls are handled in the kernel by the VFS layer.
- The VFS layer calls the corresponding function of the backing file system.
- The backing file system performs the required operation, which might be reading data from the local disk or fetching some blocks over the network if the file system is not local.
- The results are copied back to original caller by taking previous steps in

reverse.

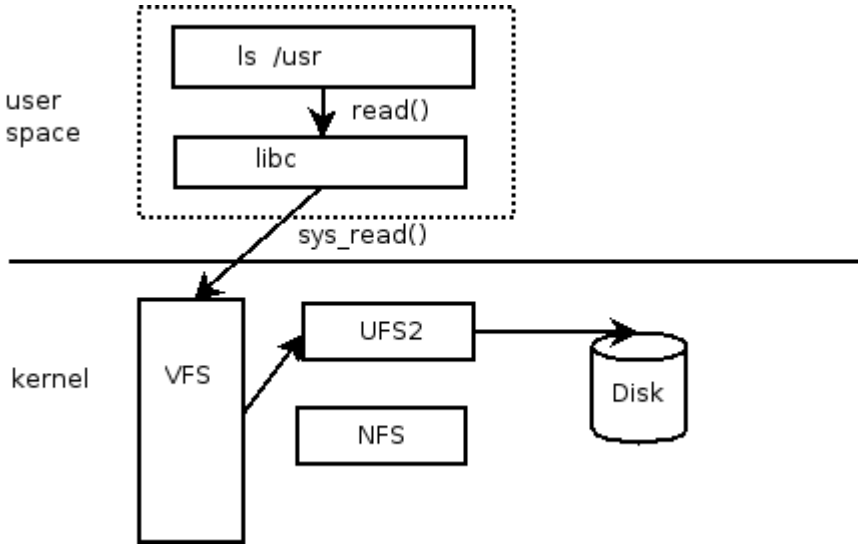

Illustration 2: Normal I/O flow in Unix systems.

3.2 What is a distributed file system?

For many purposes it is often a good idea to have files stored on one computer accessible by others. An example of such a system is a university network where you might easily have thousands of computers where each person might log in, and you want them still to see the same files on each computer. Besides being convenient to use it is also far more easier and efficient to back up this central file server than it would be to back up thousands of clients.

In early systems files were just transfered between systems with special tools like an FTP client. It was obvious though that a file system that would look to applications as a normal local file system and could do the necessary data transfers with the server transparently would be a far better solution, since it would allow us to use almost all our usual applications on top of it without having to make any changes to their source code.

Many different ideas were proposed and implemented on different levels of the system and with different semantics. Some of the implementations were on the

block layer which had good performance but lousy semantics because of the coherency problems introduced by the block caching. The other extreme was implementation right at the top of the kernel near the system call dispatch layer from where the necessary FS calls were just proxied to remote machines. This had great semantics and was easy to understand but suffered from awful performance [5]. Finally it was agreed that the best of both worlds can be achieved somewhere in the middle, where the kernel still has knowledge of the files but can also use normal buffer caches (at the *vnode* layer). This balancing between performance and semantics is the common problem in the design of all the distributed file systems (DFS). In general when you want better performance you have to relax the semantics.

Most of the current widely used DFS implementations have a simple client-server architecture where many clients are served by a single server (*NFS, CIFS*). For scalability and availability reasons it is a good idea to have multiple servers and the possibility of having a single file replicated on more than a single server. This multiple master architecture is very hard to implement because of the distributed locking and cache coherency issues that might arise. Currently AFS and Coda are the most popular of such file systems but neither is in wide use. Both of them relax the the Unix file semantics quite a bit which makes them unsuitable for the applications that really expect those, like DBMS software.

To achieve reasonable performance a network file systems use caches. In general you might cache both reads and writes. Caching the reads is somewhat easier: the main danger will be that the file has been changed on the server and your cached copy is out of date. A simple solution used by NFS is to keep the caching timeframe short so the probability of serving false data is low. Another option is to introduce the concept of *cache leases*/*callbacks* which means that the server will keep track of which client has which file and sends them invalidation notices when the master copy changes. Such a scheme is used by most of the network file systems (*AFS, Coda, SMB, Sprite*) [6].

One of the most important aspects of any distributed file system is the naming.

Ideally you do not want filenames to reveal a hint about the files physical location which is called *location transparency*. Somewhat more advanced property is called *location independence* which means that file name does not need to be changed when the files physical location changes [1]. Logical next step from having location independence is to have multiple copies of the same file called replicas on different servers for better performance, availability and fault tolerance. Since having same files on multiple servers creates additional cache coherency issues there are only few distributed file systems that have implemented replication.

Some systems like *NFS* allow you to mount remote directories into any place in the clients local directory tree. While this solution is extremely flexible it might also be very hard to administer since every client can have different resulting file system structure. Alternative to this approach is to force single global name structure that will look the same from all the client machines in the cluster. The latter approach is the one used in *AFS* based file systems.

3.3 Case studies

In order to get a better understanding of where the main implementation difficulties lie and what the most common solutions are, we will now analyze some of the specific distributed file systems. We have decided to group them on the basis of whether they are implemented in kernel, userspace or are hybrid between the two. There are of course various other important aspects that could have been used for grouping like single master vs. server-less systems, caching, failure tolerance and so on but we have found that whether a file system is implemented in kernel or userland determines most of the other design aspects.

We define kernel based file systems as those that have any component of the file system itself in the kernel and userland file systems as those that do not have any special help from the kernel. We call file system a hybrid one when it uses some kind of kernel proxy layer (*portalfs, FUSE, LUFS*) that is not specific to the file system implementation.

It is important to notice that by this definition the FS implementations that have some parts in the userland and some in the kernel (i.e *Coda*) fall into the kernel based class rather than the hybrid one.

Up until lately all distributed file systems were implemented in kernel space because the speed penalty of having to copy data continuously back and forth between userland and kernel used to be unacceptable. Only in recent years has the CPU speed of cheap commodity hardware reached levels where the performance improvements offered by the pure kernel space implementations have become marginal and bottlenecks have moved to bandwidth and network latencies [7].

3.3.1 Kernel based implementations

The main problem with implementing anything in the kernel is that you cannot use familiar libraries, mistakes will usually crash the whole OS instead of only your program, debugging is hard and you have to be aware of lots of limitations like a small stack size. All this means that usually implementing a stable file system in kernel space requires several years of development time and if you have to do network communication and synchronization with other nodes the task will become a lot harder. This is probably one of the main reasons why there have been so few kernel space distributed cluster file system implementations so far.

3.3.2 NFS

Sun Microsystems Network File System (NFS) is one of the earliest non-experimental distributed file systems and probably the most used one currently. It is also widely used in current Elion infrastructure hence we will discuss its architecture. In this text we are discussing NFS version 3 even though the version 4 that brings important caching improvements was approved recently too it is not yet widely implemented or used.

NFS has a pure client-server architecture and is stateless which allows it to be

more robust in the face of network failures. It does not have its own on-disk structure and can be rather thought of as a stacked file system that you can use to share any of your real local file systems with others. Basic idea of the architecture is that NFS client component marshalls incoming file system calls into remote procedure call (RPC) messages which are then executed on the server (Illustration 3).

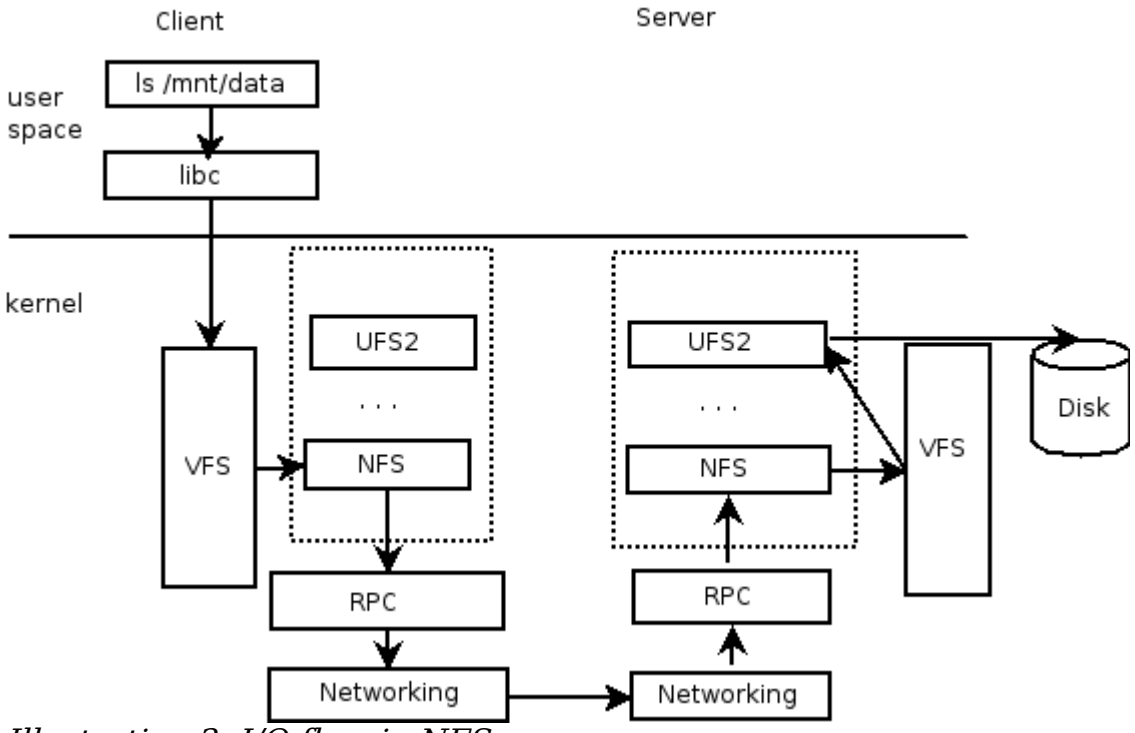

Illustration 3: I/O flow in NFS

There is a very little support for caching – basically you can cache data for reading on the client side but have to verify modification timestamps each time before using cached entry. Some implementations have decided to relax this requirement and will not verify cache entries that are accessed many times in rapid succession. While being good for performance it introduces a small window of possible cache incoherence. All the communication with the server is done with remote procedure calls (RPC) each of which carries all the necessary information for doing a single I/O request [5]. Since there is no state or much of a caching support the server never has to contact its clients, which is a huge

design simplification. Having no state avoids all the complexities of data synchronization, but on the other hand it also means the nodes have to be connected by fast network to achieve reasonable performance and locking support has to be implemented by other means.

In the light of our declared goals the main problems with such an approach are:

- Availability: when the connection to the server fails or the server crashes you cannot access your files anymore.
- Scalability: while the single server approach works fine for a small number of clients it will quickly become a bottleneck in any larger installation like a university network with thousands of nodes.
- Fault tolerance: since the files are physically located on a single server they can easily be destroyed when something happens to it (fire, earthquake, electrical surge etc).

3.3.3 Coda

Coda is a distributed file system that has been in development at the Carnegie Mellon University (CMU) since 1987. It started out as an advancement of AFS, which was developed at the CMU too. AFS in turn was heavily inspired by the design of NFS and is still in active development by the OpenAFS and Arla projects. There also exists a file system called InterMezzo which started out as an improvement of the Coda so the lineage has become quite complicated.

The original purpose of the AFS and latter the Coda project was to create a university wide general purpose distributed file system for CMU. Obviously that meant it had to scale to serve thousands of client nodes without saturating the network. The cluster had to have many servers since no single server could handle such a large number of clients and supporting server replication was essential to fulfilling the high availability requirements that this kind of environment has. They also implemented far better security models than NFS by adding support for Kerberos authentication, access lists and encryption. Coda

also supports disconnected operation which means that a client that gets disconnected because of network failures or on purpose (i.e mobile clients) can still do operations on locally cached files. Changes are later merged back to the cluster when the client joins it again. There might of course be conflicts when a disconnected client has modified a file that was modified by others too. In these cases the conflict resolving process might require human intervention.

Coda achieves high performance by ensuring that most operations will only touch the local file system. Since various studies have shown that a great majority of the file system operations are reads they decided just to fetch the whole file into local client's cache when the *open syscall* is issued. After that all the operations on that file can be done with a speed comparable to local file systems. Coda actually even uses a local file system to store cache. Write operations are sent synchronously back to server if possible, if not then they are written to local log which will be played back when client joins cluster.

One of the envisioned usages for Coda was supporting mobile computing where clients join and leave the cluster regularly, often for days. To this end they have added a concept of hoarding that allows you to mark some files that you will most often need as sticky which ensures that they will always be in cache. In addition they have a helpful program that “spies” on your usage habits and can mark the files that you will most probably need sticky for you.

Coda is almost suitable for our goals but still has couple of serious limitations:

- Files are always replicated in whole before *open* call returns [8]. While being acceptable for small files it is unusable for large files that our content will primarily consist of.
- We do not have much control over cache placement or any centrally accessible information about current placements.
- It has problems with multihomed hosts that use multiple ip-addresses on different interfaces, which is something that we need to do for network traffic optimization purposes.

3.3.4 Userland approach

Compared to the kernelspace method, the userland approach is a completely opposite method. The idea is to relax the meaning of a filesystem and implement some of the filesystem calls like *readdir(), open(), read()* etc. as a userland library that is directly linked with your applications that want to use it. This method is of course a lot simpler to get right, but you have to modify all your tools and client applications to understand the semantics of your library. This means that you cannot use the normal system tools on your filestore.

3.3.5 MogileFS

MogileFS is a pure userland file system implemented by Danga Interactive for their Livejournal project. It is basically just a set of file replication scripts written in Perl. It has a flat namespace and can support multiple methods for actual file replication. Metadata is stored in a MySQL database. A client that wants to use it has to first ask from a tracker where the required file is located and then issue a retrieve command for the file directly to the storage server.

3.3.6 GoogleFS

GoogleFS is a distributed proprietary file system implemented by Google for internal use. Because their core business requires constantly handling huge amounts of data and there were not any cost effective solutions available that could scale well enough, they decided to create their own distributed file system that would be optimized to their specific workload and scale well to a large number of servers and clients. They also decided to run it on top of inexpensive commodity hardware since it meant significant cost savings.

Google is running several GoogleFS clusters, the largest of them reaching several hundred terabytes distributed over more than a thousand machines. The main design decisions with their implications were the following [9]:

- Using normal PCs for nodes necessitates constant monitoring, error detection, fault tolerance and automatic error recovery. In other words

component failures must be considered normal rather than something exceptional.

- Most of their files are rather large, usually several gigabytes each. Because of that they decided to use 64MB as block size which is much larger than is usually used in file systems.
- Reads are mostly large and sequential meaning that a file is usually read from the beginning to the end.
- Writes are mostly large streaming appends. Random writes at arbitrary position are rare and do not have to be efficient.
- Concurrent appends are normal and must be deterministic.
- The file system and the applications running on top of it are both designed by Google so there is no need to provide a strictly POSIX compatible interface to the file system.

The architecture of the system consists of single master server containing all the metadata and many chunkservers that store the actual data. Files are split into chunks (analogous to blocks in the classical file systems).

Metadata server stores file and chunk namespaces, mappings from files to chunks and locations of each chunks replicas. Control over the replication decisions is left to chunkservers and indeed the master does not even keep the information about chunk placements in any persistent form. Instead it will ask each chunkserver for this information at the startup and whenever a chunkserver joins the cluster. Master keeps all the information in RAM for performance reasons. Important metadata changes are written to log file on the disk which is replicated to other servers for backup. Periodically snapshots of the system state are created to keep log files small. On startup the master server just reads in latest snapshot and replays and subsequent log entries.

3.3.7 Hybrid implementation

As a compromise between the two you can use a special pseudo file system that allows you to mount a userland process onto a directory in a file tree. The kernel just reflects all the I/O syscalls for this mountpoint to the userland process that handless it in whatever way it sees fit and returns the result to the kernel. The kernel then just copies the result back to the original caller (Illustration 4). This way the users of your file system see it as a normal FS and can use any programs that can work with the normal file systems and in the meantime you get all the flexibility and ease of the implementation of the userland file system.

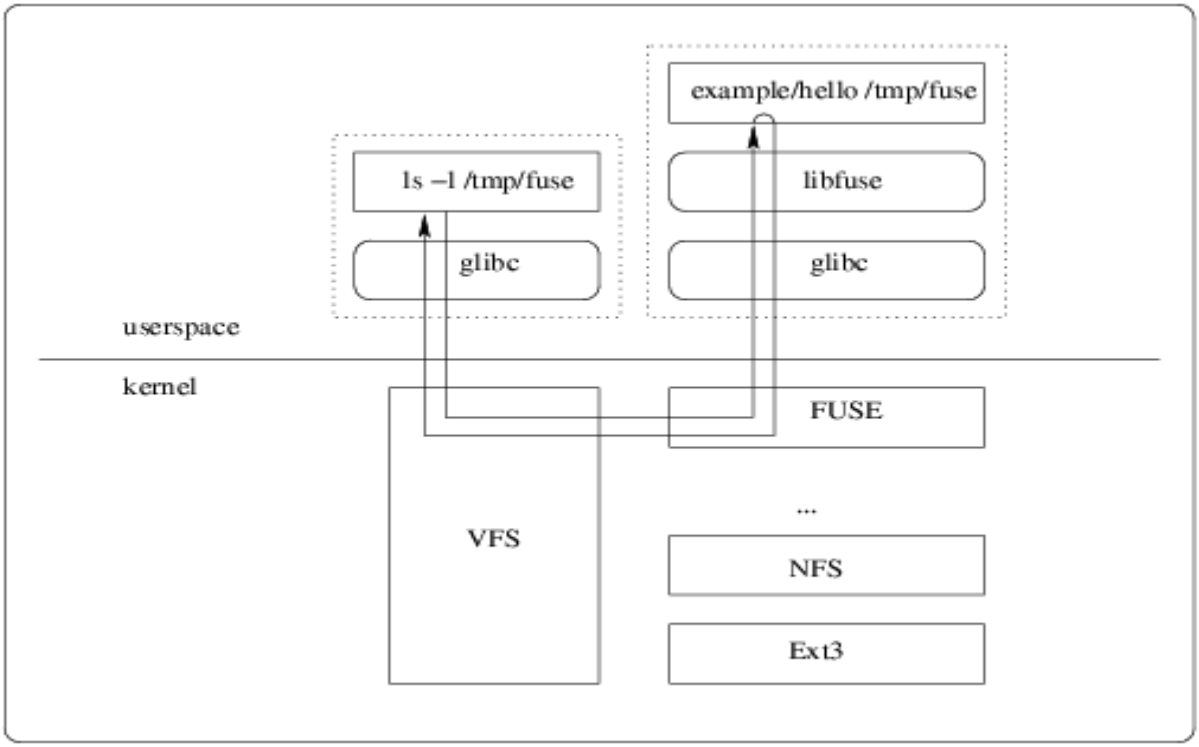

Illustration 4: I/O flow in hybrid filesystem

One of the earliest of such file systems was *portalfs* in 4.4BSD [10]. A demo function implemented on it was the ability to mount a TCP connection interface as a file system so when you wanted to open a socket to the SMTP port of some host you could do it just by opening the file */net/tcp/example.com/smtp* and use the filehandle as if it were a normal file.

Recently the same concept has been implemented in Linux as the FUSE [11] file system and was later ported to several other operating systems. FUSE seems to be rather popular with more than 50 different file systems implemented on top of it. It also has bindings for many very high level languages like Perl, Python, Haskell and Lisp.

Our own RPFS system is also a hybrid built on top of FUSE using the Python bindings.

3.3.8 TDFS

TDFS [7] is a FUSE based file system written in C. Each TDFS host is running a master and a slave daemon. The master daemon is mounted as a FUSE file system and it connects over TCP/IP to one or more slave daemons running on the same or other hosts. It is a stacked file system that can use any native file system for the real file storage backend. TDFS requires you to have all files available on all the nodes and this replication should be done by other means (with *rsync* for instance). Since all the files exist on all the systems the reads will always be served from the local copies.

While TDFS is closest FUSE based file system to our goals that we could find, it still is built for completely different purpose which makes it unsuitable for our purposes. Namely it is meant to be used for keeping content that is shared by many servers in sync when all the servers have all the data. A good example of such a workload is a farm of servers serving the same files in round-robin configuration.

4 Design Requirements

Originally we thought that we might just implement a simple distributed flat namespace file system and do all the hierarchical views, quota- and privilege management in the HTTP- and FTP daemons. However, it became clear that such an approach would cause a lot of code duplication and would be hard to maintain, so we decided to push everything into the file system layer.

With the original design goals discussed in chapter 2 in mind we formulated the following requirements for our file system:

- **Hierarchical namespace**, because it has to look like a normal file system to applications.
- The filesystem must implement **access control list** (ACL) based privileges, since our privilege separation needs are far more complicated than classical Unix user/group model can handle.
- **Quota management** is needed so that users wouldn't be able to upload too much through FTP or other protocols.
- Automatic **load balancing**, so that files that are often accessed will be replicated to more nodes.
- We have to provide **custom views** to allow users to see files shared to them by other users directly or through group memberships in their home directories.
- High **fault tolerance**: node failures should be considered a rule rather than exception since we use cheap off the shelf PC hardware.
- Some **knowledge of the** geographical **topology** of the cluster and link costs, because our cluster will have nodes in several cities and links between the cities are a lot slower than local ones.
- **Several logical files** with different owners and privileges **might refer to the same physical file** to save space when the hash of the files is

identical. This is done to save space since it is very likely that several users have identical large video files.

5 Design Decisions

Since kernel based file system projects required many years of work by teams of several full time employees, it was rather clear that writing our FS into kernel was not a viable option if we wanted to get the project done in any reasonable timeframe. The pure userland approach did not seem to be desirable since we really wanted to be able to use standard commands, system management tools and daemons on it. This left us with the FUSE based hybrid approach.

We decided that the high level design should be based on the rules of Unix philosophy [12]. The rules we considered most important, with short descriptions of their applications to our system are as follows:

- **Rule of Modularity:** *write simple parts connected by clean interfaces.* We designed to split the system up into several daemons that communicate with each other to achieve their purposes. Each daemon process has a simple clearcut purpose.
- **Rule of Separation:** *separate interfaces from engines.* For example replication daemon is built up from several internal components and the highest layers have no idea over which transport methods files will be replicated.
- **Rule of Economy:** *programmer time is expensive; conserve it in preference to machine time.* We decided to write our system fully in Python which is a very high level language allowing fast development.
- **Rule of Optimization:** *prototype before polishing. Get it working before you optimize it.* Because of the modular design we can easily rewrite parts of our system to C if optimization needs should arise, without touching parts that do not need optimization.

Since it was clear that writing something like that was still far from trivial, even with the given choices and principles in place, we also made several important technical simplifications based on our specific expected workload. Most

important ones are similar to those made by the Google file system:

- Files are rather large in most cases. 500MB is probably a reasonable mean size to expect. This means that we do not have to perform very well for small files. There is a special case where we have to serve many small files, that is handled by the optimization discussed in 6.2.1.
- Files are almost always read from start to end sequentially. Since we always need a full file there is no reason to use blocks as the replication unit and we can always do replication in full files. Another reason why blocks are often used is of course speed, since it allows you to request subsequent blocks from different peers in parallel. In our case we did not feel that the possible speed improvements would be worth the added complexity.
- Writes are almost always sequential as well, hence bad write performance at an arbitrary offset is acceptable as long as appends are fast.
- Changes (writes and deletes) are very rare compared to reads. The primary implication of this is that we could use a single metadata server without having the fear of it becoming bottleneck in any foreseeable future.
- None of our applications needs file locking, so there is no reason to implement it.
- Parallel writes to the same file should never happen. So we can just reject any writes to the file when somebody is already changing it.
- Most of the files are audio, video or executables. Since compressing these file types will not have much of an effect we will not implement it.

For the metadata storage we selected MySQL, because we already had a powerful MySQL cluster along with the management knowhow in place for other purposes and using it instead of some custom metadata server allows us to easily use some of the metadata knowledge directly in the portals that use the same

SQL server. Using SQL also made sense since it provides all the necessary referential integrity and serialization guarantees.

Obviously the single metadata server is the main potential bottleneck in our architecture but we are confident that we can push the file system to many thousand storage servers before our metadata server will become saturated, since we have gotten more than 50000 queries per second from our SQL cluster in benchmarks. Even if we should eventually run into this limit it will be trivial to add additional read-only slave servers.

Another important decision was to write the file system in userspace using FUSE through its Python bindings. This decision is by far the most important of all, since it probably saved us years of development time. Using such a high level scripting language may sound really awful performance-wise, but probably it will not be much of a problem since the file system code will mainly deal with resolving paths to *inode* IDs, fetching remote files and handling privileges. Actual I/O is mostly done directly from the backing file system to network using *sendfile()*. Even in the cases where we have to send files out in the write loop the context switch and copy overhead is probably dwarfed by network latencies.

One of the most important things was to understand that performance and scalability are very different things, the file system might not perform very well on one server, but if it is easy enough to add a couple of more servers to the system it will probably be cheaper in the long run than spending time on coding it into the kernel.

6 Implementation

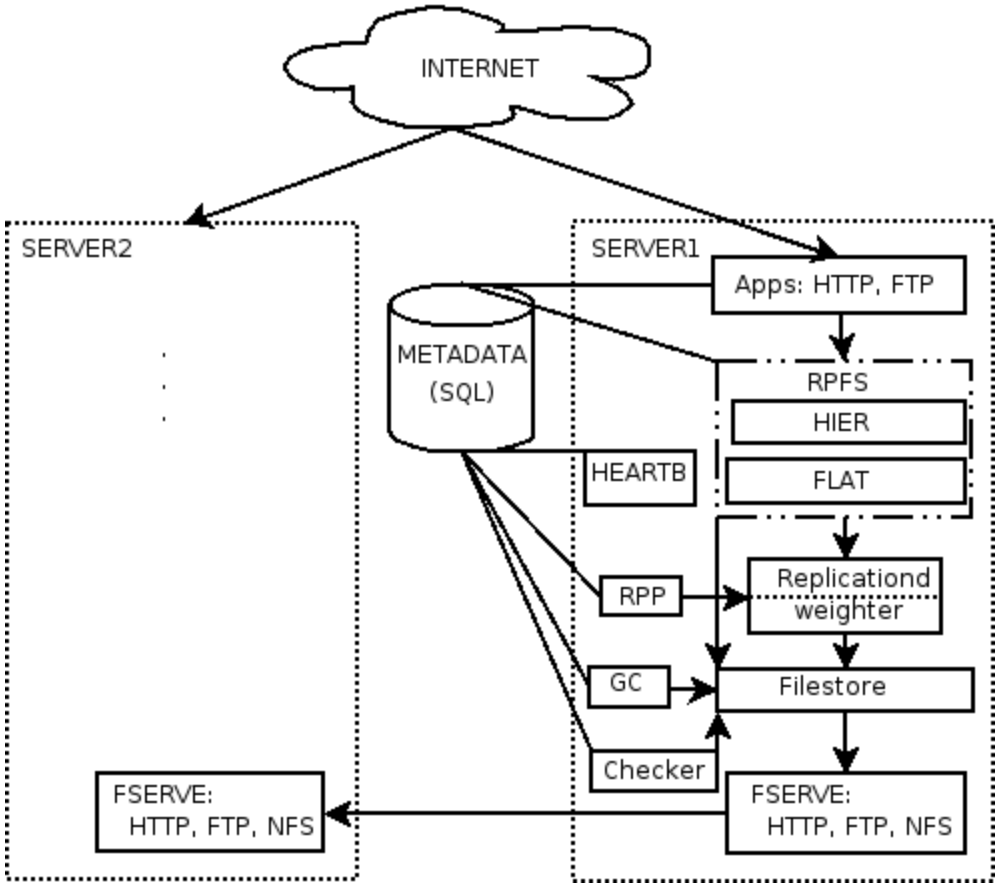

Illustration 5: detailed view of the RPFS cluster

We have a number of separate processes running on their own*:*

- The *RPFS* component that is built on top of FUSE and consists of two stacked layers (FLAT and HIER).
- *Replicationd* (*RPFSD*) that handles the actual file replication. It also has a separate thread called *weighter* that is responsible for calculating preference ordering of the peers.
- *Periodic replication checker* (RPP) that periodically checks if there is anything new in the cluster that we should replicate.
- *GC* - Garbage Collector that deletes files that are no longer referenced

from the metadata.

- *Heartb* – the heartbeat deamon that periodically writes information about the server load and other performance metrics to the metadata server.
- *Checker* – periodically calculates the checksum for local files and compares them to the metdata in order to detect corruptions.
- *Metadata* – a central data storage where file system metainfo can be kept.
- *Filestore* – Storage for locally replicated files, on a native FS like UFS or EXT3.
- *Fserve* – a method for serving local filestore to all our other servers in the cluster (HTTPd, NFSd etc.).

6.1 Metadata layout

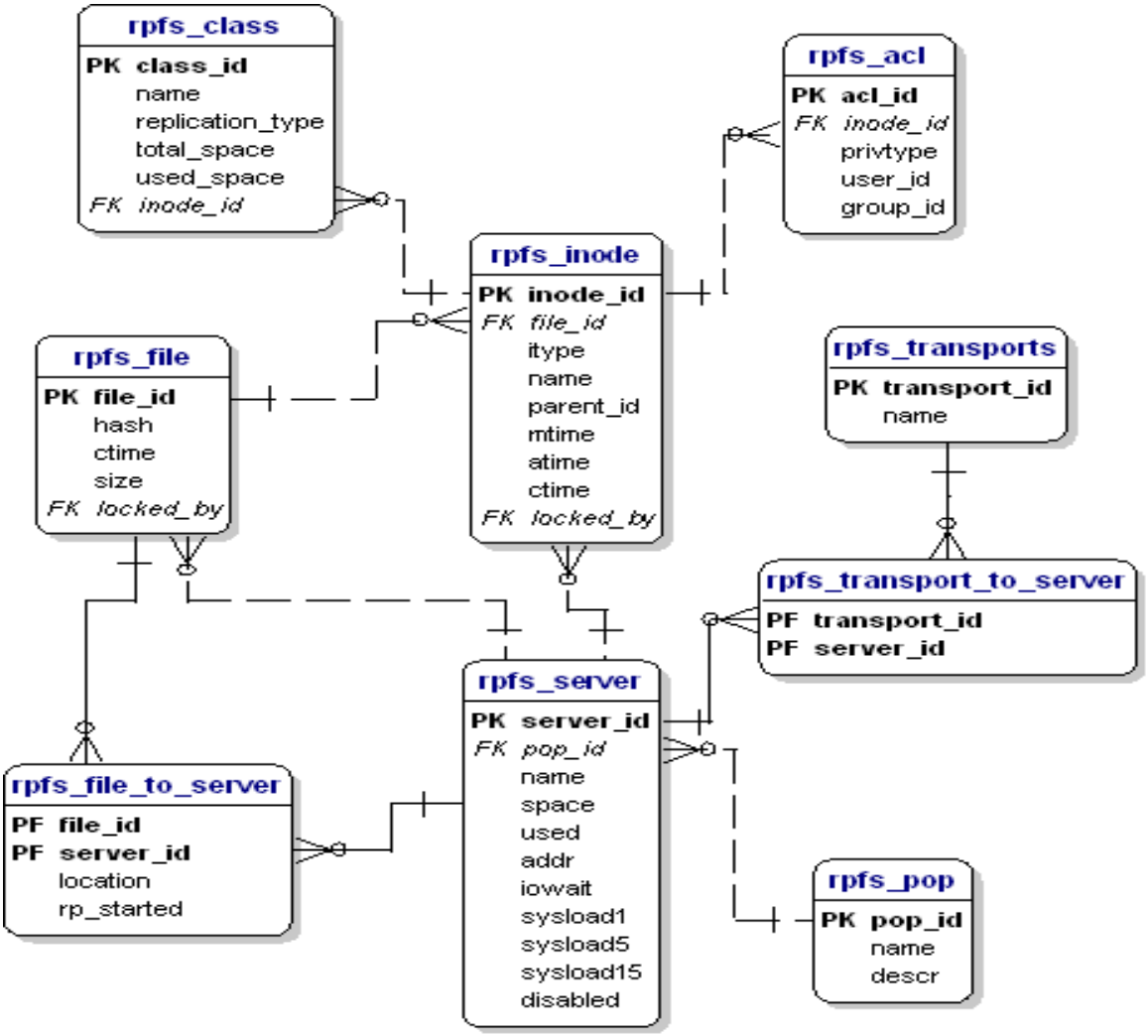

Illustration 6: Metadata SQL schema

6.1.1 RPFS_CLASS

This table describes the storage classes which are somewhat analogous to the mountpoints in classical file systems. When we mount RPFS to a storage server we have to specify which storage class we want to mount.

Attribute	*Datatype*	*NOT NULL*	*Description*
class_id	Int	+	Primary key

Attribute	*Datatype*	*NOT NULL*	*Description*
name	varchar(255)	+	Name of the class.
replication_type	enum(full, demand)	+	How should files be replicated in this class.
total_space	Int		Estimate of total space.
used_space	Int		Estimate of used space.

6.1.2 RPFS_POP

Describes Points Of Presence (POPs). It is used to group servers that are located physically in more or less the same location. In our production system POPs are different cities. These groupings are useful for getting an overview of the cluster topology and are used in replication decisions, i.e. we always prefer to replicate a file from the server that is located in the same POP as we are if possible.

Attribute	*Datatype*	*NOT NULL*	*Description*
pop_id	Int	+	Primary key
name	Varchar(255)	+	Name
descr	Varchar(255)		Optional description

6.1.3 RPFS_TRANSPORTS

Describes the different replication transport methods (HTTP, NFS, FTP, etc.).

Attribute	*Datatype*	*NOT NULL*	*Description*
transport_id	Int	+	Primary key
name	Varchar(255)	+	Name
preference_order	tinyint		Used to specify the default transport method preference ordering. For example it can be

Attribute	*Datatype*	*NOT NULL*	*Description*
			used to tell that HTTP should be always preferred to FTP if possible.

6.1.4 RPFS_SERVER

Describes a single physical server.

Attribute	*Datatype*	*NOT NULL*	*Description*
server_id	Int	+	Primary key
pop_id	Int		Foreign key
name	Varchar(255)	+	Name of the server
space	Int		How much total disk space is available on the RPFS filestore mountpoint (bytes).
used	Int		How much space is currently used by the filestore (bytes).
addr	Varchar(255)	+	IP address of the server. IPv4 or IPv6.

6.1.5 RPFS_TRANSPORT_TO_SERVER

One to many mapping table between *rpfs_transports* and *rpfs_server* tables.

Attribute	*Datatype*	*NOT NULL*	*Description*
server_id	Int	+	Foreign key
transport_id	Int	+	Foreign key

Attribute	*Datatype*	*NOT NULL*	*Description*
preference_order	Tinyint		Can be used to override default transport preference order for this specific server.

6.1.6 RPFS_FILE

Describes the actual physical files that are stored in the native FS.

Attribute	*Datatype*	*NOT NULL*	*Description*
file_id	Int	+	Primary key
hash	Varchar(255)		Hash of the file. Currently we use SHA256
ctime	Datetime	+	File creation timestamp.
size	Int	+	Filesize in bytes.
storageclass	Int	+	Foreign key

6.1.7 RPFS_FILE_TO_SERVER

One to many mapping table from *rpfs_file* to *rpfs_server*.

Attribute	*Datatype*	*NOT NULL*	*Description*
file_id	Int	+	Foreign key
server_id	Int	+	Foreign key
rp_started	Datetime		When this file is currently being replicated to this server this field contains the replication start timestamp. When file is completely replicated it's set to

Attribute	***Datatype***	***NOT NULL***	***Description***
			NULL.

6.1.8 RPFS_INODE

Provides a hierarchical namespace on top of the flat “block layer” (*rpfs_file*).

Attribute	***Datatype***	***NOT NULL***	***Description***
inode_id	Int	+	Primary key
file_id	Int		Foreign key
itype	Enum('F','D','M')	+	*Inode* type.
name	Varchar(255)	+	Object name.
parent_id	Int		Foreign key (parent *inode*)
mtime	Int		Last modification timestamp.
atime	Int		Last access timestamp.
ctime	Int		Object creation timestamp.

Mtime, atime and ctime are stored as Unix timestamps.

6.1.9 RPFS_ACL

Describes the access control lists for *inode* objects.

Attribute	***Datatype***	***NOT NULL***	***Description***
acl_id	Int	+	Primary key
inode_id	Int	+	Foreign key
Privtype	Enum('R','W')	+	Type of privilege.

Attribute	*Datatype*	*NOT NULL*	*Description*
user_id	Int		
group_id	Int		

6.1.10 RPFS_MSGBUS

Used as a journal of changes that servers use to keep their RPFS cache states in sync.

Attribute	*Datatype*	*NOT NULL*	*Description*
id	Int	+	Primary key
msgtype	Char	+	What was done with the object. Currently the allowed values are D (Deleted) and C (Changed).
evt_ts	Timestamp		Event timestamp.
mobj	Int		*Inode* ID

6.2 File system

RPFS really consists of two separate file system layers stacked on top of each other. *HIER* is the top layer that provides a hierarchical namespace and does privilege and quota management. *FLAT* is the lower *RPFS* layer that only has a flat namespace without any privilege checks. FLAT is in turn backed by a native file system like *ext3, ReiserFS* or *UFS2*.

6.2.1 Flat

Provides the basic flat view of the cluster. It is basically a single folder

containing all the files that exist in the cluster. Filenames are *inode_id* attributes from the *rpfs_inode* table. *FLAT* also handles the communication with *replicationd* if necessary so that *HIER* does not really have any knowledge of replication issues whatsoever.

The flat view might be useful by itself for optimization purposes if an application knows exactly what *inode* it wants and the content does not need privilege checking. In that case we can skip the path resolution completely by servicing objects directly from RPFS storage mounted with flat view. The main examples of this kind of workload are public galleries and multimedia sharing sites. For these applications we also generate links to specific backend servers that are known to have the file instead of using a generic cluster address. This way the servers never have to send out replication requests which cuts down access time considerably which is important when we are serving many small files (for example thumbnails of photos) instead of few large ones which was our intended primary workload.

This flat view optimization possibility is actually of course just a nice side effect of using SQL for metadata storage as opposed to the *inode* tree used in normal file systems.

6.2.2 Hier

Hier is a layer built on top of the flat namespace, and provides a hierarchical structure that users are familiar with. In addition, *Hier* does all the privilege and quota checking. We cannot use the normal Unix uid/gid model for our security needs since the users really just do not exist in system, so we have to get privileges from the path and rely on the servicing applications chrooting the users correctly.

Similarly to classical Unix file systems our internal Hier layer structure is built around the *inode* abstraction that represents any object in the filesystem. Internally the *inode* object is backed in metadata storage by *rpfs_inode* table. In current implementation we have 3 types of *inodes*: files, directories and

metanodes.

Files

Each file object is backed by one entry in the *rpfs_inode* table that has its *itype* attribute set to *F.* Each file references a single actual raw file object in the *rpfs_file* table. Multiple *inodes* might reference the same raw file which is an optimization for the case when many users have the same files. When write is done through one of the multiple *inodes* referencing the same raw file the system will just perform copy on write (*COW*) transparently and create a new raw file ID for this *inode*.

Directories

Directory is an *inode* that has its *itype* attribute set to *D*. Directories can contain files, metanodes or other directories.

Metanodes

In order to be able to allow FTP we had to create a home directory for each user where he could be chrooted on login. In addition to the users private files, galleries and other such resources it also has to contain all the resources that are shared with the user either directly from another user or indirectly through a group mechanism like an intranet. While it might sound deceptively simple, it has proven to be the most complicated part of our system. We saw only three possible implementation possibilities:

- Use of symbolic links (*symlinks*) to reference various shared directories from the users home directory. It would have been the simplest to implement but keeping track of hundreds of thousands of *symlink*s all over the system would have been really hard to manage in the long run. Also since *symlink*s cannot be followed outside *chroot*s it would have been infeasible security wise too.
- Mount shared directories to users home directories via so called loopback

or *null* mounts which allow you to mount one point of directory tree into another. While this will work fine inside *chrooted* environment it still is hard to manage and keep tract of. Also we feared that OS probably cannot handle hundreds of thousands *mountpoints* in a single system gracefully.

- The solution that we chose was to create special *inode* types in our *RPFS* that are similar to directories but know how to find out what shared directories they are allowed to show in themselves.

These special directories are called metanodes and are *HIER* objects that are implemented by subclassing *MetaNode* object which in turn subclasses *Inode* object. In general metanode can get the content that it returns from any source, not just the *rpfs_inode* table like normal directories.

For example we have a UsersMetanode in our FS that selects all the users from our users table so when you ask for a directory listing you will see a directory with name of each user that exists in the SQL. Each of the entries in the UsersMetanode is again a metanode called UserMetanode which represents home directory of a single user and contains hardcoded list of other metanodes like IntranetsMetanode which contains a single IntranetMetanode for each intranet user belongs to (Illustration 7).

While there are no restrictions on where the directory list returned by metanode comes from, there still are some restrictions on what objects the metanodes can contain and what operations are allowed on them. These limitations are mainly caused by the fact that metanodes jump out of our normal directory hierarchy and do not have real *inode_id*'s so we cannot reference them from the parent field of the *rpfs_inode* table. Currently the limits are as follows:

- You cannot create or delete a metanode from the file system with usual operations. They can only be operated with by special file system management tools.
- Metanodes can only contain directories or other metanodes.

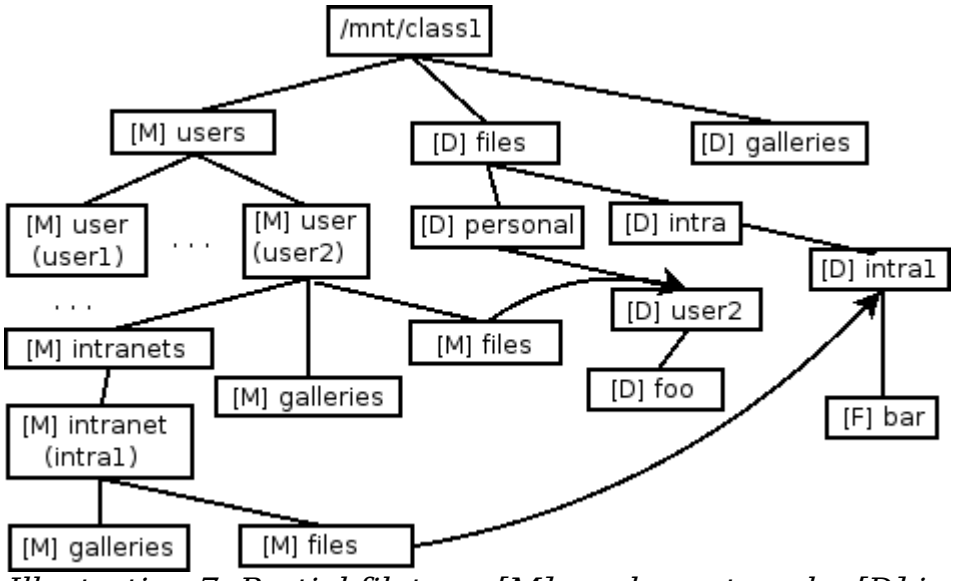

Illustration 7: Partial filetree. [M] marks metanode, [D] is a directory and [F] is a file

6.2.3 Caches and cache coherency

One of the most common operations in almost any file system is name lookup which is the process of resolving a path to an *inode* ID. Without any caching we would have to ask the metaserver for the *inode* ID of every path element. This would quickly overload the server and would be rather slow because of the communication delays. In order to offset the name resolving costs most file systems have a *namecache* that directly caches path-to-*inode* mappings for recently requested paths in a hash table or some other data structure that allows fast lookups. The RPFS processes implement a *namecache* by keeping the *inode* object tree in RAM. When the tree grows over the configured size, a garbage collector will traverse it and delete the least recently used nodes.

The main problem with caching metadata - especially in a distributed environment - is that we have to be notified of changes if we want to be sure that we are not serving old content to users or even worse - serving content to users that no longer have privileges to access it. Our cluster does this through the *rpfs_msgbus* table which resembles a journal in the normal file systems. When

there is a change in *inode* information, a new record specifying the affected *inode* ID and the operation type is also added to this table. The garbage collector thread in each RPFS process constantly polls this table to see if there are any new entries with their ID greater than the last one seen. If so then it remembers the new last record ID and purges all the affected *inodes* from the cache. The *metaserver's* cronjob deletes entries from the *msgbus* table that are older than 24 hours, daily. Obviously we still have a small inconsistency window between the change event and the moment when the *inode* is actually removed from the caches but usually it should be around ten seconds which we consider acceptable.

We also implemented a hash based cache that contains the direct pathname to *inode* object mappings but had to disable it because of the complexity involved in finding out which entries should be purged. This difficulty arose from the fact that there can be multiple paths leading to the same *inode*.

6.2.4 Locking

In order to ensure consistency of our directory tree and file contents we have to protect some operations with locks. Currently we have a *locked_by* fields for this purpose in the *rpfs_inode* and *rpfs_file* tables which are a foreign keys referring *rpfs_server.id* entry of the lock holder.

When a write operation is attempted on a file handler for the first time or a new file is created we will lock corresponding *rpfs_file* and *rpfs_inode* entries to ensure the file is not changed on other servers at the same time. When *close()* is called for this filehandle *RPFS* updates the timestamp, checksum and size fields and releases the locks. The locks are semantically considered write locks so servers can still serve *read()* requests for locked files and even do replication. Any write requests for the locked files from other hosts will block.

Locking is also needed when multiple files entries are collapsed into one by integrity checker as discussed in section 6.7. In that case we will first lock both *rpfs_file* entries and then all the *inodes* that reference them. Then *rpfs_inode*

entry is changed to reference the *rpfs_file* entry that will remain after the collapse and locks will be released in the reverse order.

When *write()* is called for a file that is referenced by multiple *inodes* we will first copy the entire file on the backing store, create a new locked *rpfs_file* entry for it and change our *rpfs_inode* entry to reference it instead of the new one.

6.2.5 Object removal

When a program issues the *unlink()* syscall for a unlocked file object we will just delete the corresponding entry from the *rpfs_inode* table. Actual deletion of the file and *rpfs_file* entry is left to the garbage collector process discussed in 6.4. If the *inode* is locked we will return *EIO* error to the caller.

When a directory object is removed with the *rmdir() syscall* we can just execute a simple SQL delete statement without worrying about locking since SQLs referential integrity will ensure the necessary consistency.

6.3 Replication Daemon

All the real replication work is done by a separate daemon called RPFSD. The client processes that want some file to be locally replicated request it from RPFSd over a local Unix domain socket with a simple HTTP like protocol. RPFSd is solely responsible for deciding where to get the file from and possibly selecting the actual transport method for getting it if multiple possibilities are present.

RPFSD is designed to be transport agnostic - with each server we store a list of supported transport methods in the order of preference. Transport methods might vary from a pure userland protocol like FTP to network filesystem mounts like NFS. Currently the supported transport methods are HTTP and NFS.

Clients can request file replications over a special protocol called RPCOM that closely resembles HTTP. Using this protocol clients will ask RPFSD to replicate files and are notified of the state changes in replication. Responses are either

final or transitional. A final response means that RPFSD has finished servicing the replication of the given file and no further notifications about it will be sent unless the client asks for the replication again. Transitional message signals important change in replication state, for example when replication has been started we return a transitional message to the client so that the client may start writing out the file to its callers.

Example of the data flow in the RPFS cluster when a non-local file called *foo.avi* with *inode_id* 97 is requested is shown in the Illustration 8.

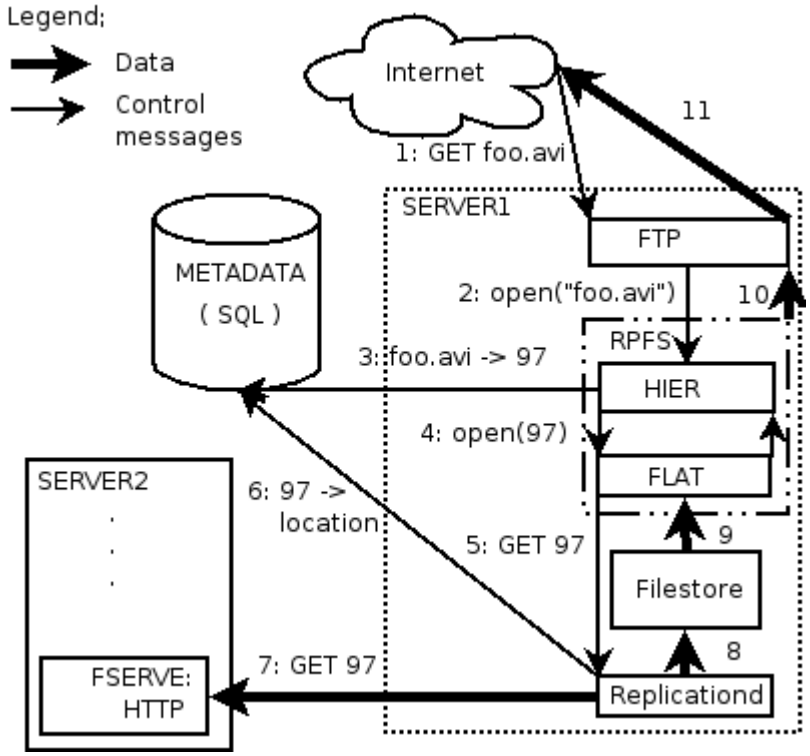

Illustration 8: Data flow in RPFS when non-local file is requested

Currently there are two clients on each datanode that communicate with RPFSD: the RPFS file system itself and the periodic replication checker (RPP).

One slightly complicated issue we faced with RPFSD design was the question how to block RPFS reads on I/O when our replication is in progress and a client is reading faster than RPFSD can replicate. As discussed previously Coda solves

this problem by blocking *open* calls until file is completely replicated. Because our primary workload consists of huge files such a solution is unacceptable.

Basically we saw three feasible solutions:

- Ignore the issue and hope such situations occur rarely enough. The client will just get a partial file in this case.
- Add a special *ioctl* to the kernel that can be set on files to signal that the file is currently under replication and readers should block when the end is reached instead of returning EOF.
- Force RPFS to send out files that are under replication in the write loop, instead of using *sendfile*.

Kernel implementation would have been obviously fast but also far more complicated and it would have required maintaining our patches against the new versions of the kernel. It would also have required extra work if we wanted to use other operating systems besides Linux. So with the Rule of Economy in mind we decided to use a slower but easier to implement method of adding a replication state aware I/O loop to RPFS.

6.3.1 Structure

The logical design of the RPFSD consists of:

- *A comlink* interface that handles RPCOM communication with clients.
- A backend replication worker thread spool that handles the actual file replication work.
- Two queues that act as a communication layer between the two.
- A weighter thread that calculates link costs to peers.

We decided to split the communication interface and workers up so that the code logic would be cleaner. Having multiple worker threads might be good for scalability on multicore systems that are already widely used too.

The *comlink* interface parses the requests from the client and puts them into the central request dictionary and *workq*. The free worker threads take requests from the *workq* and do the real replication: they use *request_queue* to communicate back the changes in replication state. The *comlink* is responsible for taking the requests from the queues and communicating the state changes back to all the interested clients. When the *comlink* communicates back a state change that is final it also removes the request from the request dictionary.

6.3.2 RPCOM protocol

Clients ask the *replicationd* to replicate the file with “GET file_id“ command.

RPFS responds with a status code followed by the *file_id* and possibly explanation of the status. Code semantics are similar to the ones HTTP protocol uses i.e. 200 means success, 3XX are various temporary errors, 4XX are permanent errors and 5XX are various unexpected conditions.

Each message ends with the decimal value 13 (Unix newline). The format of line is

code file_id [message] where

- code - integer in the range 200-599
- file_id - 64bit integer > 0
- message - optional additional information

Currently the following codes are defined

Code	***meaning***	***Final?***
200	replication of file finished	yes
201	replication of file started (file should be visible on backing storage)	no
202	replication of file is accepted and queued	no
304	temporary replication error for example when daemon cannot connect to any peers that have the copy	yes

Code	*meaning*	*Final?*
404	permanent replication error - metadata storage does not know anything about such a file	yes
405	permanent replication error - file metadata exists but nobody really has it	yes
500	something unexpected occurred during replication	yes

6.4 Garbage Collector

RPFS does not send out file deletion commands to data nodes when the file is removed from the metadata server. Instead we implement a lazy deletion where file or directory removal only deletes the corresponding entry from the *rpfs_inode* table so the file will not be reachable through *HIER* anymore.

Each data server runs a garbage collector process (*GC*) that periodically checks if:

- there are raw files that are replicated on our server according to the metadata and are not referenced by any *inodes* anymore, in which case we can delete that file from the *file_to_server* mapping table and remove it from the local filestore. If it is the last replication entry for a given file it will also remove the raw file record from the metadata (*rpfs_file* table).
- There are files on our node that are over replicated and the last access time is older than the configured value.

GC can also delete files that are still referenced if the server is running low on space and the replication constraints of the given file allow it. Usually GC runs in user configured intervals but users can ask it to do a sweep right away by sending the USR1 signal.

6.5 Performance measuring daemon

One of the daemons called *Heartbeat* periodically writes some performance

metrics about the system into the central metadata storage. Currently we store the mean system load values for the last 1,5 and 15 minutes and the iowait value.The *weighter* threads of all the servers use those values in link cost calculations in order to prefer less loaded servers.

6.6 Weighter

The purpose of the weighter thread is to periodically calculate link costs to our peers to ensure that we use the most optimal replication paths. It runs as a separate thread inside the replication daemon.

The cost for each host is determined by the following equation:

$$linkCost = \frac{sysload5 * ioload}{cpusInTheSystem} + networkSpeed$$

Where *ioload* and *sysload5* are the corresponding values from the *rpfs_server* table for that peer and *networkSpeed* is the time it took to fetch a single static 512kB test file, which is measured by the weighter thread itself. When a file is available from multiple hosts we always prefer the one with the lowest link cost.

6.7 Integrity Checker

In order to detect corruptions caused by hardware problems as early as possible we run a daemon that does regular sweeps over all the locally stored files, calculates their checksums and compares them against checksums from central metadata storage. If mismatch is found then alert is logged and offending file is deleted.

Integrity checker is also responsible for collapsing multiple *rpfs_file* entries that have the same checksum into single entry. This is done simply by updating corresponding *rpfs_inode* records to reference the *rpfs_file* entry with the highest replication count.

6.8 Filesystem tester

Tester is a distributed file system testing script. It is not of course strictly part of the file system, but we consider it essential to have a simple command at hand for testing most of file systems capabilities. This way we can catch bugs caused by new changes right away which simplifies the development process considerably.

It consists of two process groups called readers and writers. Both groups contain a predefined number of child processes, usually ten. Writers sleep for random time intervals and create random directories and files in the file system. They sometimes also delete and modify existing files. Reader processes traverse the file system randomly and sometimes try to read random files.

In general this testing script creates far greater load on the cluster than we ever expect to see in a production system and has helped us to uncover various bugs, locking issues and corner cases both in our own code and the OS.

7 Technical Architecture

7.1 Hardware

Our actual system has currently five RPFS nodes located in three cities and each city is a different POP. Servers are built from cheap desktop components with the following specs:

- AMD X2 3200 (dualcore 2,3Ghz 64bit CPU)
- 4G RAM DDRAM (400Mhz)
- four 500GB SATA disks
- two 1Gbit/s network interface integrated on the motherboard

Disks are currently in software RAID5 configuration so we have around 1,5TB of storage available on each node. This allows us to survive loss of at least 1 disk in each of the nodes without data losses. Of course we could have done without the RAID because the RPFS should be able to survive the loss of a single node, but we decided to play it safe while the system is still young and bugs might exist in failover scenarios.

Metadata is stored on the SQL cluster that has two servers with the following hardware:

- four 2.4Ghz 64bit AMD Opteron CPUs
- 12G RAM (400Mhz)
- Storage is attached over storage area network (Fibre Channel) from EMC CLARiiON CX300

7.2 Software

Storage servers are running following software:

- **Debian Etch** Linux with 2.6.20 kernel
- **FUSE** 2.6.5
- **Python** 2.4
- **PyDog** application server (ver. 0.3). This is a proprietary server implemented by Elion Enterprises Ltd. and used by our *hot.ee* portal. It's used on storage nodes for serving content over HTTP.
- **Vsftpd** 2.0.5. Used for serving content over FTP.
- **PAM MySQL** authentication module used by vsftpd
- **lighttpd**

ReiserFS is used as a backing filesystem.

Metadata cluster is running:

- **Suse Enterprise 9**
- **MySQL** 4.1.14 with InnoDB tables.

The metadata cluster is not really specific to RPFS and runs several other databases as well.

7.3 Structure

Each RPFS data server serves content to clients over FTP and HTTP protocols using vsftpd and PyDog servers respectively. Users manage their file privileges and quotas from our hot.ee portal which actually uses the same SQL database as RPFS metadata storage so the management is done directly using SQL queries without any intermediate interface in between. One of the network interfaces on each RPFS data node has a public IP address and the other one is in private VLAN. All the metadata and replication work is done over the private network. We use HTTP as our current replication transport so we also have lighttpd webserver running on all the private interfaces for that purpose.

One of the more interesting optimizations is that each datanode has two public

IP addresses, one of which is the same for all the nodes and when client tries to open connection to it he is routed to one of the nodes in the closest POP. Because of this clients in several cities can open connections to the same IP address but still get connected to the closest server. The unique IP address is used in cases where we actually want to get the specific node – for example *hot* portal might serve gallery application and generate links to thumbnails as URLs to several specific backend servers to distribute load between them.

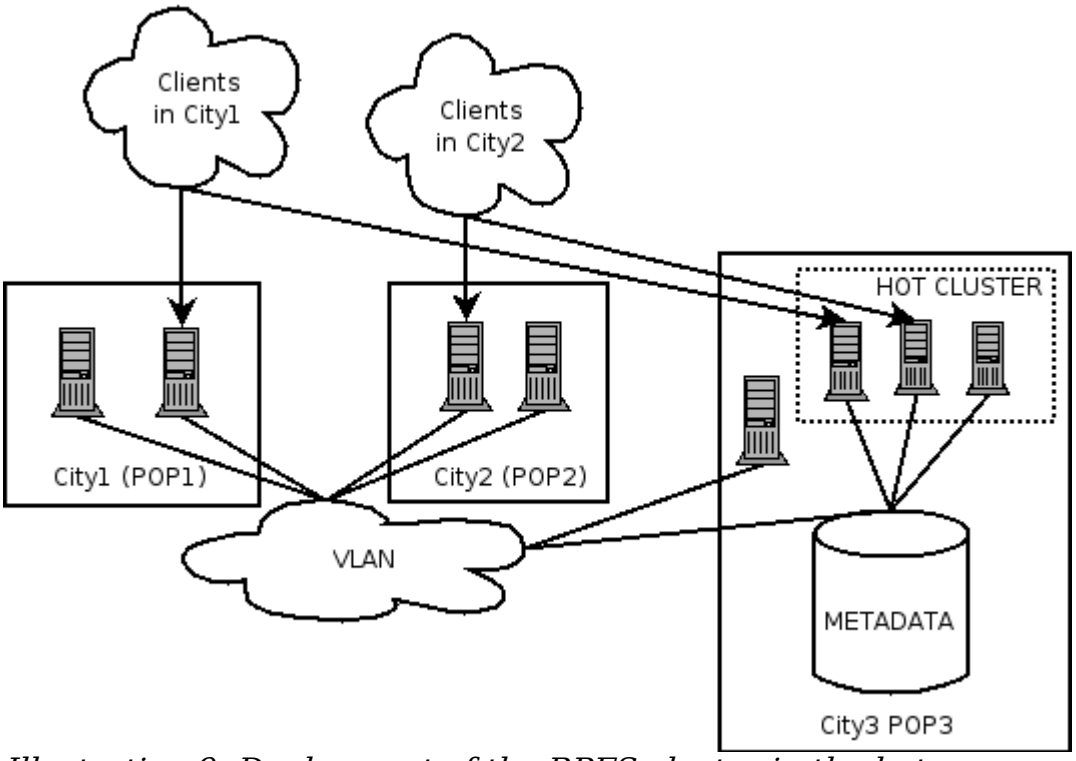

Illustration 9: Deployment of the RPFS cluster in the hot.ee environment

8 Summary

The purpose of this thesis is to propose a design and implementation of a distributed file system that could be used as a storage platform for various current and future services at Elion Enterprises Ltd.

In this work we have discussed the goals set for this project and the reasoning behind them, which made up the first part. The second part of the thesis explained some of the core concepts and terms. After that we discussed various existing distributed systems in relation to our goals.

The central part of the thesis, which is at the same time its main result, describes the design and implementation of our own file system called *RPFS.* Essentially our file system is a stacked one, running on top of the FUSE layer that allows us to provide a normal file system interface to programs while running in userland. We have made and discussed several important design simplifications which allowed us to implement the system in a reasonable timeframe. Currently the implementation is in the early testing phase.

The author has learned a lot about the issues that arise in distributed file systems and how they are solved in several current implementations. The field of distributed file systems has proved to be a interesting one and this thesis has only managed to scratch the surface.

9 References

1. Silberschatz, Abraham; Galvin, Peter; Gagne, Greg *Applied Operating System Concepts* John Wiley & Sons,2000 0471365084
2. Love, Robert *Linux Kernel Development, Second Edition* Pearson Education,2005 0672327201
3. Silberschatz, Abraham; Galvin, Peter; Gagne, Greg *Windows XP supplement chapter* 2002
4. *Files 11* http://en.wikipedia.org/wiki/Files-11 28.05.2007
5. McKusick, Marshall Kirk *The design and implementation of the FreeBSD operating system* Pearson Education Inc.,2004
6. Braam, Peter J; Nelson, Philip A.; *Removing bottlenecks in distributed filesystems: Coda & Intermezzo as examples.* Proceedings of the 5th Annual Linux Expo, pages 131-139,1999
7. Voras, Ivan; Žagar, Mario *Network Distributed File System in User Space* 2006
8. *Whole file caching in Coda* http://www.coda.cs.cmu.edu/misc/wholefile.html 28.05.2007
9. Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung *The Google File System* 2003
10. McKusick, Marshall Kirk *The design and implementation of the 4.4 BSD operating system* ,1996 0201549794
11. *Homepage of the FUSE project* http://fuse.sourceforge.net 28.05.2007
12. Raymond, Eric S. *The Art of UNIX Programming; 1st ed.* Addison-Wesley Professional,2004 978-0131429017

10 Resümee

Tänapäevased meediarohked veebirakendused nõuavad aina suuremate andmemahtude talletamist ja serveerimist, kusjuures kriitilised on ka nõuded veatolerantsusele ja skaleeruvuse potentsiaalile. Sarnase vajadusega seisis silmitsi ka Elion ja selleks otstarbeks otsustati luua oma hajus failisüsteem.

Antud töö kirjeldaski Elioni ootusi taolisele süsteemile, disaini, selle taga olnud põhjusi ja tehnilise rakenduse arhitektuuri. Muuhulgas analüüsiti ka mitmeid mujal maailmas samalaadsete probleemile lahendamiseks loodud hajussüsteeme, millest me oleme oma arhitektuuris ka paljuski eeskuju võtnud. Väljapakutud failisüsteemi disain ongi selle diplomitöö põhitulemuseks. Kirjutamise hetkel on kirjeldatud failisüsteem juba ka realiseeritud ja on parajasti varajases testi faasis.

Autor õppis töö käigus palju uut erinevate hajusfailisüsteemide loomisel tekkivate probleemide ja nende tüüpiliste lahenduviiside kohta. Selgeks sai ka, et tegu on veel arengus oleva alaga, kus on palju lahendamata probleeme ja seega ka huvitavaid väljakutseid edaspidiseks teadustööks.