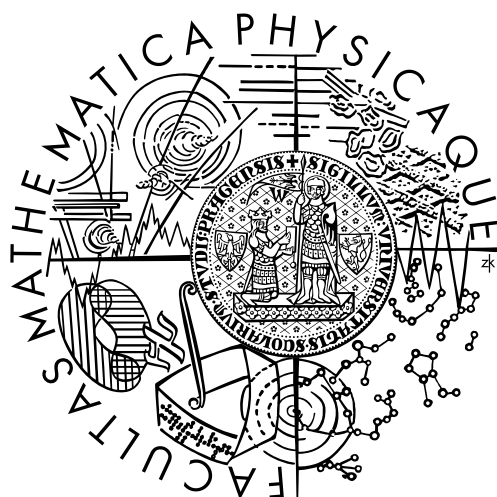Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Josef Zlomek
Shared File System for a Cluster
Department of Software Engineering
Supervisor: Ing. Petr Tůma, Dr.
Study Program: Computer Science

I would like to thank my supervisor, Ing. Petr Tůma, Dr., for his valuable advice.

I declare that I have written this master thesis on my own and listed all the used sources. I agree with lending of the thesis.

Prague, August 8, 2004   Josef Zlomek

Contents

Název práce: *Sdílený systém souborů pro cluster*
Autor: *Josef Zlomek*
Katedra (ústav): *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *Ing. Petr Tůma, Dr.*
e-mail vedoucího: *petr.tuma@mff.cuni.cz*

Abstrakt: *ZFS je nový distribuovaný systém souborů, který umožňuje skupině počítačů připojených do sítě mezi sebou transparentně sdílet některé adresářové stromy. Každý počítač může pracovat s adresářovými stromy, které se nacházejí na jiných počítačích, přímo přes síť, nebo si může některé adresářové stromy cachovat na svém souborovém systému, aby se snížila zátěž sítě. Tato lokální cache se automaticky synchronizuje.*

Uživatel může používat soubory a adresáře cachovaného adresářového stromu dokonce během doby, kdy se jeho počítač nemůže připojit k počítači, který poskytuje tento adresářový strom. Lokální změny se reintegrují, až se počítači podaří znovu se připojit. Pokud několik počítačů provedlo kolidující změny cachovaných souborů nebo adresářů, souborový systém konflikty odhalí. Konflikt je v souborovém systému zobrazen jako adresář, který obsahuje jednotlivé verze souboru. Konflikty se vyřeší tím, že uživatel verze porovná a smaže nežádoucí.

Klíčová slova: *distribuovaný, systém souborů, provoz bez připojení*

Title: *Shared File System for a Cluster*
Author: *Josef Zlomek*
Department: *Department of Software Engineering*
Supervisor: *Ing. Petr Tůma, Dr.*
Supervisor's e-mail address: *petr.tuma@mff.cuni.cz*

Abstract: *ZFS is a new distributed file system, which makes it possible to transparently share directory trees by a group of network nodes. Each node may access the directory trees located on other nodes remotely, or it may cache certain directory trees in its local file system to reduce network traffic. This local cache is automatically synchronized.*

The file system enables a user to access and modify files and directories of the cached directory tree even while it cannot connect to the node that provides the directory tree. The local changes get reintegrated when the node succeeds in connecting again. If several nodes change the cached files or directories in a conflicting way, the file system detects the conflicts and represents them as directories, which contain the particular versions of the corresponding files. The conflicts are resolved by the user by comparing the versions and deleting the unwanted ones.

Keywords: *distributed, file system, disconnected operation*

Chapter 1

Introduction

Users of a group of computers that are connected to a network usually need to share applications and other files. They usually want the files to be located in one logical place so that they would not have to remember where specific files are stored.

The solution that is commonly used is to use a file server, which exports a large storage capacity via a network file system. However, the file server is not cheap, and is often overloaded so the access speed is slow. Therefore, a better solution is required.

The users might want to logically join the unused parts of their computers' disks into a single file system, and thus create a huge storage capacity that is spread over many computers. In order to do so, they employ a distributed file system.

The users who use laptops often take their unfinished work with them, for example when they go to a conference or on a business trip, so that they could continue working when they have time, for example in a train or airplane. Such users want a simple way to take the important files that are stored in the shared file system with them, and to reintegrate the changes made during the trip to the shared files. Copying the files manually to the local disk and comparing them when the user returns is far from simple, and the user can easily omit important files. The user would be happy if the file system itself provided a transparent way to take the files with the laptop, allow the user to work with them, and automatically reintegrate the modifications when the laptop connects to the network again. Such a feature is called *disconnected operation*.

When the user connects the laptop to the network via a modem, he or she would not be happy if the file system overloaded the connection. Therefore, the user wants the file system to support *mobile operation*, i.e. to adapt to the connection speed.

1.1 Goals

When a group of nodes is connected by a fast network, it would be nice if there was a file system that would make it possible to share subtrees of the nodes' directory trees among all nodes. Because several nodes may be laptops, which can be disconnected or connected via a slow link, the file system should support disconnected and mobile operation.

Although Coda [14] can be used for this purpose, it is not well suited for it. The main reason is that Coda expects the group of nodes to be divided to servers and clients. The servers only enable the clients to access the file system and do not access it themselves, and the clients always cache the files on their local disks.

The purpose of this thesis is to design a file system that would address this issue and to write its implementation. The goals of the file system are described below.

Each node may provide several directory trees, called *volumes*, to the other nodes. The volumes should be mounted in one directory tree, and it should not be obvious from the path on which node the volume is located. The nodes may access the volumes provided by the other nodes directly through a network, or they may transparently cache[1] data of certain volumes on their local disks. Accessing volumes located on the same node should be effective with respect to disk space.

The nodes cache the volumes for the two main reasons. The first reason is to speed up access to files that are not changed too often. The second one is to support *disconnected operation*, which allows a user to use the files when the node is disconnected.

If the volume is cached on a node, the local cache should be automatically synchronized when the node is connected via a fast link. But when the node is connected via a slow link, the file system should limit network traffic to save the bandwidth, i.e. it should support *mobile operation*. If the file system is not able to automatically synchronize the files because of a conflict, the conflicting versions of corresponding files should be visible in the file system. It should be easy for the user to choose the right version of the file and thus resolve the conflict.

The file system should also be relatively easy to install and use, therefore it should not depend on other software packages where it is not necessary.

---

[1] The node chooses whether and from which node it wants to cache the volume in its local configuration.

1.2   Structure of the Thesis

Chapter 2 gives a brief overview of several existing distributed file systems and summarizes why none of them fulfills the goals of this thesis completely. It also analyzes different approaches to solve the fundamental questions of a design of a distributed file system and proposes solutions that fulfill our goals.

Chapter 3 describes the design of the file system from the high-level view. Alternatives are discussed where appropriate, and the reasons are given for why one of them was chosen while the others were not.

In chapter 4, the important or interesting parts of the implementation are described in more detail. Alternatives are compared where appropriate.

Finally, chapter 5 concludes the thesis. It shows that all goals of the thesis were met and highlights the main contributions.

Chapter 2

Distributed File Systems

This chapter gives a brief overview of several existing distributed file systems and summarizes why none of them fulfills the goals of this thesis completely. It also analyzes different approaches to solve the fundamental questions of a design of a distributed file system and proposes solutions that fulfill our goals.

2.1 Existing Distributed File Systems

2.1.1 LOCUS File System

LOCUS [12] is a complete distributed system. It includes an interesting distributed file system, whose main features are described below.

The file system consists of file groups, which are the subtrees of the whole directory tree. The file groups are replicated by having multiple physical containers on different nodes for each logical file group. One of the physical containers is the primary one, and the others are secondary ones. The physical containers may have different sizes.

A client node may read data from any node that contains an up-to-date version of the requested object. The modifications are send to the node holding the primary physical container. This node then sends a message, which contains new version number and a list of modified blocks, to the other physical containers. The secondary physical containers then retrieve the modified blocks from the primary one.

The file system allows processes to work with the file groups even if the network is partitioned. In this case, a new primary node is selected in each partition. When the partitions connect to each other again, the non-conflicting changes are merged. However, the literature does not mention

what happens when the changes are in conflict.

LOCUS enforces Unix semantics of sharing, i.e. that all writes are visible to all readers. It employs read tokens and write tokens. A node has to have the corresponding token to be able to perform an operation. Several nodes may hold read tokens, but only one node may hold a write token while no other nodes hold read tokens.

The file system supports *network transparent pipes*, i.e. processes on different nodes may communicate via a pipe as the processes on one node in Unix. Devices are network transparent too.

The file system in LOCUS is a nice file system. However, it is not clear what it does with conflicting changes.

2.1.2   Google File System

The Google File System [4] was designed by Google to meet its needs. It is designed to be high-performance for large files that are usually read or appended sequentially in large streams. It expects that components often fail. However, it does not implement a standard API such as POSIX.

Files are split into chunks, which are 64 MB large, and these chunks are stored in a local file system of *chunk servers*. The chunks are replicated on multiple chunk servers. Metadata, like name space, a mapping from files to chunks and chunk locations, are managed by a replicated *master server*. The master periodically checks whether the chunk servers are online. The master also manages the replication. It makes placement decisions, creates new chunks and other things.

When a client wants to read a file, it sends the file name and offset to the master and the master replies with a chunk identifier and a list of chunk servers that the client should contact. The client then interacts with a chunk server that is closest to it.

When the client wants to write to the file, it asks the master what chunk server holds the lease, such a chunk server is called the *primary*, and which chunk servers hold the other replicas. The client pushes data to all chunk servers and then sends the write request to the primary. The primary writes the data to the chunk and forwards the write request to all other replicas, which then write the data too. After all replicas have acknowledged the write, the master replies to the client.

The Google file system expects that nodes are very loosely coupled so it was not designed to support disconnected operation. No single node may access any file of the file system. Its design is very different from what would be required to fulfill our goals.

2.1.3   Global File System (GFS)

The Global File System [16] targets environments that require a large storage capacities and high performance, such as scientific computing. GFS is a file system for architectures that contain storage devices attached to the network. These storage devices are not owned by any node.

Data of the file system are located on the storage devices and nodes access these storage devices directly. The file system data are striped across several devices. There are *device locks* attached to devices, which are used to synchronize the access to the device. The locks are acquired in increasing order to prevent deadlocks. To improve performance, GFS caches data on the storage devices, which together with device locks avoids the need to maintain cache coherence on the nodes.

GFS is a good file system for distributed applications running on nodes that work with the same data. The target usage of GFS is different to the proposed usage of ZFS so it is no wonder that GFS does not address our goals, such as disconnected operation.

2.1.4   Petal and Frangipani

Petal and Frangipani are two parts of a distributed file system. Petal [9] presents a collection of virtual disks to users, such as file systems or databases. Frangipani [18] provides the high-level view of the file system on top of the Petal virtual disk.

The Petal virtual disk is spread over multiple servers and divided into logical blocks, which have two physical replicas on different servers. The block can be read from any replica. A client preferably writes to the primary replica, which propagates the write to the secondary replica before a reply is returned to client. If the primary server is down, the write goes to the secondary server. The block on the secondary server is marked so that Petal could know what blocks have to be written to the primary server when it starts again. Petal does not expect that the servers get partitioned so it does not care about conflicts.

Frangipani supports Unix semantics, i.e. the changes made to a file or directory on one server are immediately visible to all others. In order to ensure this, a *distributed lock service* is employed. Frangipani makes use of read-write locks, and locks the whole files and directories instead of blocks. A read lock allows a server to read the file and cache it (in memory). A write lock allows the server to write to the file in addition to reading and caching. When the lock service detects conflicting lock requests, the current holders of the lock are asked to release or downgrade it. When the server is asked

to release the read lock, it has to invalidate the corresponding cache parts. When the server is going to release or downgrade the write lock, it has to flush dirty pages first. In case of releasing the write lock, the cache must be invalidated too.

Because the data of the file system are spread over multiple nodes, neither node can access a part of the file system on its own. Therefore, Frangipani is another file system, which does not support disconnected operation.

2.1.5   Zebra

Zebra [6] combines ideas from log structured file systems and RAID disk arrays. It batches small independent writes into larger sequential writes, stripes this data stream from one client across multiple servers, and uses parity to improve availability. Because of the usage of parity, Zebra survives the loss of any single storage server.

The metadata are maintained by a *file manager*. As soon as a client has written a stripe, it sends the information about the stripe's contents to the file manager. When the client wants to read the file, it gets the metadata of the file from the file manager, gets the corresponding stripe fragments and reconstructs the file from the fragments.

The fragments are read-only. When the client wants to modify the file, it has to write new data to a new stripe fragment. A *stripe manager* is responsible for cleaning old fragments. If the old fragment still contains valid data, it is read and written to a new fragment.

Zebra is an interesting file system. However, its reliability is lower than Petal because it uses less redundant information. The nodes cooperate to provide the access to the file system and the data is spread over all nodes, therefore Zebra does not support disconnected operation too.

2.1.6   Coda

Coda [14] is a file system, which has many interesting features, for example it supports disconnected and mobile operation.

The file system consists of volumes. The volume is provided by a group of replicated servers, called Volume Storage Group. The replication is ensured by clients, which read from one server but write to all of them. The clients also initiate solving of an effect of temporarily disconnection of several servers by comparing the versions and starting resolution on the servers.

The clients use a write-back cache for the contents of the volumes in a proprietary partition on their local disks. This cache is used by the kernel and controlled by a cache manager, which synchronizes it with the servers.

Coda supports *disconnected operation* [8] by using the local cache even when the client is disconnected from the servers. The local changes are reintegrated to servers when the client connects again.

Sharing semantics in Coda is session semantics. The changes made by a node are propagated to server when the file closes. It is necessary to detect conflicts and to provide the user facilities to repair them.

Coda has a powerful security. It employs Kerberos to authenticate users and Access Control Lists to check access rights.

Coda does not fulfill our goals completely because it does not allow the volumes to be accessed without caching. It also is not designed for the case when the client and server are on the same node. The conflict representation is not very good and resolution of conflicts requires a special tool.

2.1.7   InterMezzo

InterMezzo [2, 3, 5] was designed to have almost all features of Coda but far less code. It supports disconnected and mobile operation, and uses session semantics and ACLs.

In contrast to Coda, InterMezzo uses the local file system on both clients and servers. It consists of a kernel component called Presto and a server process called InterSync. Presto is a layer between VFS and certain local file systems such as ext3, JFS, and ReiserFS. It tracks file system changes in a Kernel Modification Log, and when a log page is full, it is passed to Inter-Sync to reintegrate it. InterSync uses the HTTP protocol for communication between the client and the server.

InterMezzo does not allow the client to access the volume remotely so it does not fulfill our goals too.

2.2   Fundamental Design Decisions

2.2.1   Data Location

In any file system, it is necessary to decide how the data of the file system will be organized. This section analyzes various aspects of data location in a distributed file system.

Organization of Data among Nodes

First, it is necessary to decide whether the file system should be monolithic or split into smaller parts, which are usually called filegroups [12] or volumes [14].

If the file system is monolithic there usually is not a node that could hold all its data because it would have to have an extremely large storage capacity so such a file system does not look like it consists of one large volume. The monolithic file systems look like they are built on top of virtual disks [16, 6, 9], which are spread over many nodes. Therefore, no node contains a complete group of files that are related to each other, for example a subtree of the file system directory hierarchy. Therefore, this type of file system is not applicable for our goals, especially for disconnected operation and providing the directory trees by each node.

When the file system is split into relatively small volumes[1], the whole volumes can be located on nodes. Each volume may be located on a different node and each node may contain several volumes. The volumes may be replicated on several nodes independently on other volumes. Therefore, the use of volumes is suitable for the proposed goals.

Location of Metadata among Nodes

Another aspect of data location is whether the corresponding metadata should be stored on the same node that contains the data, or whether the metadata should be managed by special nodes.

When the metadata is located on separate nodes, as on the master server in the Google File System [4] or in the Directory Service of the Amoeba file system [11], the file system cannot support disconnected operation because each node needs both data and metadata to be able to access the file.

In order to support disconnected operation, it is necessary to store the metadata of the files on the node containing the data, as in Locus [12]. Moreover, when the node itself manages the metadata, like version numbers, of the files located on the node, updating the metadata is more efficient.

Storing Data and Metadata on a Node

Finally, the file system has to select the way how the data and metadata are stored on a node. The file system may either manage the data in an own structure, or in a local file system.

The first option is to design an own structure similar to local file systems. The main advantage is that this structure is able to hold all necessary metadata. On the other hand, this approach duplicates the effort put into implementing existing file systems and usually the result is not as fast as the

---

[1]Relatively small volume means that the whole volume is small enough to be stored on one node so it still may be ∼10 GB large.

well-optimized local file systems. Most distributed file systems organize the data on each node in this way.

The other option is to store the data and metadata in an existing file system. When the distributed file system consists of volumes, each volume may be located in a different local file system that is selected to meet the special needs of the volume, for example certain file systems are optimized for small files. The distributed file system also benefits from the caching done by the underlying file system. The use of local file system is also more stable because the local file systems are well tested. Moreover, when the metadata get corrupted, it is still possible to use the data via the local file system. On the other hand, the distributed file system usually requires more metadata than the local file system so the additional metadata have to be stored in auxiliary files.

Although both solutions may be used for our goals, the use of the local file system is better because of the reasons described in the previous paragraphs.

2.2.2 Replication

Replication means that there are several copies, called *replicas*, of all or just certain files on different nodes of the distributed file system. The file system has to provide a method to synchronize the replicas. Replication improves availability and reliability. It also improves performance by decreasing load of servers and network traffic.

Replication will be analyzed for the distributed file systems that consist of volumes because this approach meets the proposed goals. Replication in monolithic distributed file systems works similarly with one difference. The unit of replication is not a file or directory but a data block.

The simplest way of replication is the *primary replica* model. One replica is the primary replica, the others are the secondary replicas. All modifications are performed on the primary replica that "pushes" them to the secondary replicas. An alternative is that the secondary replicas are only notified that the version of the file has changed, and eventually which blocks have changed too, and the secondary replicas "pull" the blocks from the primary replica. When an unreachable node gets connected, it compares the version of its replica and the primary replica, and eventually fetches the file or invalidates own replica. The main disadvantage is that no files or directories can be modified when the primary replica is unreachable.

Another method of replication is the *weighted voting* model. This model allows the files to be accessed even if several replicas are unreachable. In order to make this possible, it requires a node to get enough votes from the other nodes to be allowed to access the file. When the node wants to read

the file, it has to get votes from a group of nodes whose size is at least a read quorum (Q_r). Each vote also contains the version of the file. The node them reads the newest version that was sent in the votes. Similarly, the node needs a write quorum (Q_w) of votes to be allowed to write to the file. The node performs the write operation on all the nodes that have sent the vote. The read and write quorum have to be chosen so that $Q_r + Q_w$ is greater than the number of replicas. This forces that the newest version of the file is among the Q_r replicas so the node reads the file from the node that holds the newest replica. The file cannot be modified by two different network partitions because at least $max(Q_r, Q_w)$ votes are necessary to allow the modification[2]. It is not even possible to create a new file because a directory has to be modified.

The previous models are not suitable for the proposed goals because they do not support disconnected operation. It is necessary to find other methods how to replicate files.

Disconnected operation is already supported by several file systems. The most utilizable one is Coda. Volumes in Coda are replicated on several servers that make up the Volume Storage Group. Clients read data from one of these servers but have to get version numbers from all servers. The reason is that when several servers were offline but are connected again, the clients have to choose the server with the newest version. The writes from clients go to all servers. The client caches the volumes in its *permanent cache* and this cache is used while the client is disconnected from the servers.

This approach with one simplification is suitable for the proposed file system. Because one goal of the thesis is that the volume is provided by one node to other nodes, there is just one server instead of the Volume Storage Group. Other nodes access the volume remotely, or cache the volume from this primary location and use this cache while they are disconnected, as in Coda. It does not matter that the server is not replicated. When the server breaks down, it might be possible to select a node that caches the whole volume to be the new server.

2.2.3 Sharing Semantics

When several replicas exist in the distributed file system or when clients cache data in memory or on disk, it is necessary to define semantics of sharing, i.e. the semantics of simultaneous read or write access by several clients.

The simplest semantics is *no semantics*. The nodes use their cached

---

[2]$S = Q_r + Q_w > N > 0 \Rightarrow max(Q_r, Q_w) = max(Q_r, S - Q_r) \geq S/2 > N/2$, where N is the number of replicas. There might be at most one such a large partition.

information for an unlimited amount of time. Nothing is guaranteed about the time when changes made by one node reach other nodes. Of course, this semantics is not very useful.

Weak semantics is slightly better but still not good enough. The client may use the cached information only for a limited time. After this time, this information cannot be used and has to be retrieved again. The modifications are being written back regularly. This sharing semantics is used by NFS[3], version 3.

Unix semantics is commonly used semantics. This model states that write operations are atomic and their results are immediately visible to readers. Although it is easy to implement this model in local file systems, it is more difficult to do so in distributed file systems. In order to guarantee exclusivity of writes, the file system may employ tokens. The node has to hold a read token or a write token to be able to read or write the file, respectively. If the node does not have the token, it must get one. If the write token is requested and other nodes have read tokens, the tokens are recalled and the write token is granted. If either token is requested and another node holds the write token, the token is taken away and the node that had the write token propagates the modifications that it has made.

Coda uses *session semantics*. The changes made by the node get visible to the other nodes after the file is closed. When two nodes have the file open for writing at the same time, the second close does not overwrite the file on the server but creates a conflict instead. The advantage in comparison to Unix semantics is better performance when one node is modifying the file while other nodes are reading it.

Completely different semantics is used in Amoeba. The files in Amoeba are immutable. The user can only create a new file and replace the old one by the new one in the Directory Service. The old file is still used by the clients that have this file open. However, it is not effective to change small portions of the file when using Amoeba semantics.

Session semantics is the best for the proposed goals. It would be also possible to use Unix semantics but performance would be worse because of the following reasons. When the node is not allowed to use the file, it has to ask the server to grant this privilege to the node. When read/write or write/write sharing occurs, it would be necessary to update the file on the other nodes after one node modifies the file.

---

[3]Network File System.

2.2.4 Summary of Selected Solutions

To summarize, the ZFS File System will have the following features.

1. The file system will consist of volumes.
2. The metadata will be on the same nodes as the corresponding data.
3. The data and metadata on a node will be stored in a local file system.
4. The nodes will cache certain volumes on their local disks.
5. The nodes that provide volumes will not be replicated.
6. The file system will use session semantics.

2.3 Comparison of Distributed File Systems

Table 2.1 on page 19 compares the mentioned distributed file systems with respect to the aspects analyzed in the previous sections. The meaning of columns and their possible values are described below.

File system type may be either split to *volumes*, whose files are located on one node and may be replicated or cached, or *monolithic*, i.e. the files are spread over all nodes according to decisions of the file system.

Metadata location shows whether the metadata are located on *separate nodes*, or on the *same node or device* that contains the corresponding data.

Data location describes whether the user files are stored as separate files in the *local file system*, or in a *proprietary* partition structure. Data location in the Google File System is *combined* because the user files are split into chunks and these chunks are stored in the local file system.

Server replication shows whether the servers are replicated, and which method of replication is used. The file systems use a *primary replica* model, replication similar to *RAID* arrays, or *explicit* replication.

Permanent caching means whether the node that does not hold a replica caches the data on its disk. Caching may be *mandatory*, i.e. everything what the node wants to access must be cached, or *optional*, i.e. the node may access the files remotely.

Sharing semantics provided by the mentioned files systems may be either *Unix* semantics, or *session* semantics.

Disconnected operation is supported by the file systems that are marked with '+'. These file systems support mobile operation too.

File System	FS type	Metadata location	Data location	Server replication	Permanent caching	Sharing semantics	Disconnected operation
LOCUS FS	volumes	same node	proprietary	primary	−	Unix	+
Google FS	monolithic	separate	combined	primary	−	Unix	−
Global FS	monolithic	same device	proprietary	RAID	−	Unix	−
Petal + Frangipani	monolithic	same node	proprietary	primary	−	Unix	−
Zebra	monolithic	separate	proprietary	RAID	−	?	−
Coda	volumes	same node	proprietary	explicit	mandatory	session	+
InterMezzo	volumes	same node	local FS	−	mandatory	session	+
ZFS	volumes	same node	local FS	−	optional	session	+

Table 2.1: Comparison of Distributed File Systems

Chapter 3

Design of the ZFS File System

This chapter describes the design of the ZFS file system from the high-level view. Alternatives are discussed where appropriate, and the reasons are given for why one of them was chosen while the others were not.

3.1 Volumes

The file system consists of volumes. A volume is a directory tree that is accessible by other nodes and may be cached by them. If the volume is provided by the node or is cached by the node, it is located in a directory in the local file system of the node. The volume must be placed in one local file system so that it is possible to link or move files between any two parts of the volume.

3.1.1 Volume Mount Points

The volumes are mounted to one directory tree, which gets mounted to node's file system hierarchy at a time. There are two possibilities where the volumes could be mounted.

The first option is to mount each volume right under the root of the file system as a directory whose name is the name of the volume. This is simple and does not need special data structures to describe the mount points except the list of volumes. But this is a rather limiting solution.

The solution used in ZFS is more flexible. Each volume may be mounted in any path from the root of the file system. The parts of the path to the volume mount point need to be represented by *virtual directories*. The user cannot do any modifying operations in the virtual directories. If volume `X` is mounted under a mount point of volume `Y`, the files or directories of

volume `Y` that have the same names as the corresponding virtual directories
are overshadowed by these virtual directories and cannot be accessed.

An example of virtual directory tree is in figure 3.1. The mount points
of volumes are the virtual directories `config`, `corba`, `ZFS`, `volX` and `volY`. If
there is a file or directory `more` in root of the volume with the mount point
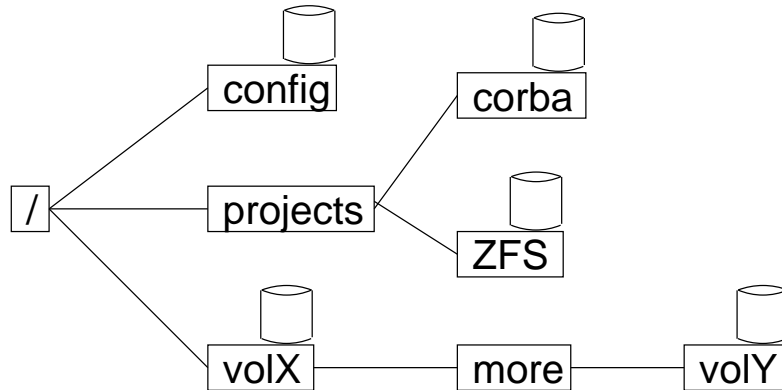`volX`, it is overshadowed by the virtual directory `more`.

Figure 3.1: Example of Virtual Directory Tree

Another question is whether all volume mount points or just the mount
points of volumes provided by connected nodes should be visible in the file
system.

If volume mount points of volumes provided by disconnected nodes were
not visible, it would not be possible to make the node connect to the corre-
sponding volume provider by doing any file system operation. Therefore, the
file system would have to provide a special tool that would cause the node
to connect, but this would not be a good solution.

So the whole virtual directory tree is visible in the file system. When
the local node wants to make a file system operation with the volume mount
point and the node providing the volume is disconnected, the local node tries
to connect to that node. If the volume provider is online, the connection is
established and the operation is performed on the volume root, otherwise
just the virtual directory is accessed.

3.1.2 Volume Hierarchy

The model with a primary location of the volume and other nodes caching the
volume or accessing it remotely, which was chosen in the previous chapter,
can be easily extended as follows. Each node may choose whether it wants to

access the volume by talking to the node providing the volume or to another node, such a node is called a *volume master*. Although the node's volume master may be any node, it should be a node that provides or caches the volume. It does not make sense for a node that does not cache the volume to be a volume master of other nodes.

The volume master relation forms a volume hierarchy (see figure 3.2, A → B means that node A is the volume master of node B). The root of the hierarchy is the node that provides the volume, i.e. the node that holds the primary replica. The hierarchy may, of course, be different for each volume. The benefit of the hierarchy is that all nodes do not interact with the volume provider to update their caches or to access the volume remotely, so the workload is handled by multiple nodes.
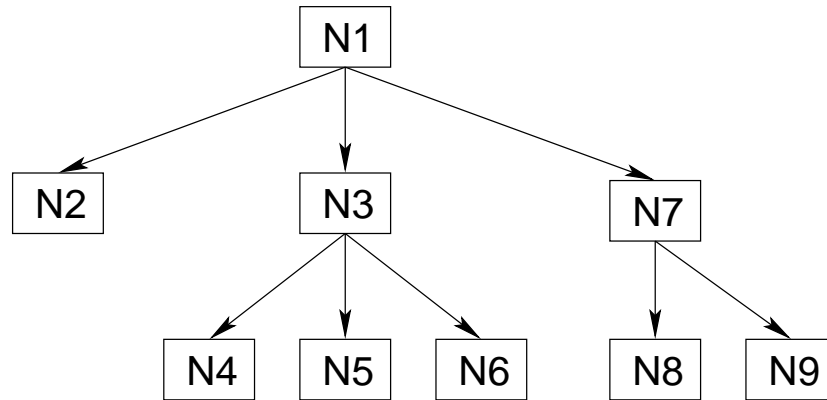
Figure 3.2: Volume Hierarchy

3.2 Accessing the Volumes

There are three distinct types of volumes from the view of a node – volumes provided by the node, volumes accessed remotely and volumes cached on the local disk. These types are accessed differently by the node as described in the following sections.

3.2.1 Volumes Provided by the Node

A node may provide several volumes to the other nodes. It means that the primary replica of each such a volume on the node's local disk. When the node is reading or modifying files or directories on such a volume, the

file system is accessing the appropriate files or directories in the local file system. It also updates its additional metadata when modifying operations are performed. Although the files of the volume are stored in the local file system, it is not a good idea to change the files via the local file system because the metadata would not be updated and thus the nodes that cache the volume would not detect the changes.

3.2.2 Volumes Accessed Remotely

The node does not hold any data of such volumes on its disk. All the files and directories of these volumes are accessed remotely. Every file system operation is invoked on the node's volume master using the remote procedure call mechanism. Therefore, no problems with synchronizing the files between the local node and the volume master arise. If the node cannot connect to the volume master, an error is returned to the user.

The remote access to the volume is supposed to be used when the volume contains files that are often accessed by at least two nodes at the same time while at least one node is modifying them, or when the node does not have enough space on its disk to cache the volume, or when the node simply does not want to cache the volume.

3.2.3 Volumes Cached by the Node

For each volume of this type, a cache is maintained in the local file system of the node. This cache contains a subset of volume's directories and files. The cached volumes support *disconnected operation*.

The cached volume could be accessed in two different ways.

The first option is to use the local cache only when the node cannot connect to its volume master, and otherwise to access the volume remotely and update the local cache for the case the node gets disconnected. However, this would require special handling according to the connection status. The local cache would not improve performance and would be good only for disconnected operation.

The option used in ZFS is to always access the local cache and try to synchronize the corresponding file or directory before doing the requested file system operation (see section 3.4). This would increase the performance and decrease the network traffic compared to the first option. The connected and disconnected operations differ only in the fact that the local cache is not synchronized when the node is disconnected. Therefore, no special handling of these two cases is required.

Both of these options require the metadata to be updated so that it would be possible to detect whether the file has changed, and to maintain a modification log (see section 3.4.1) so that the changes made to the local cache could be reintegrated. Similarly to the volumes provided by the node, it is not a good idea to change the cached files via the local file system.

The caching should be used for the volumes that contain files that do not change too often and the user wants to use those volumes while the node is disconnected.

3.3 Access rights

Every file system should provide a way to specify the access rights of files so that the read and write access can be limited to a subset of users.

The easiest way to specify the access rights in Unix environment is to grant rights to the owner of the file, to the group of the owner and to the others where the owner and the group is a system user or group, respectively. The user and group IDs have to be the same on all nodes accessing the file system, but this is not the common case. On the other hand, this solution does not need any additional information because the access rights are stored in the local file system.

A better option is to define file system users and groups, specify a mapping between the node's users and groups and the file system ones, and describe the access rights similar to the previous case. This requires the list of users and groups and the mapping for each node to be explicitly stored, for example, in the configuration of the file system. This solution is currently used in ZFS because it is still simple but more flexible than the previous one.

The best solution would be to use the Access Control Lists (ACLs), which enable to specify a list of entities (users or groups) that are allowed to access a given file. The access rights are attached separately to each entity in the list. However, it is more complicated to store the ACL in the file system because it has a variable length, and to check the permission using the ACL. Moreover, the users have to be authenticated to the file system to increase the security, for example by Kerberos. According to [1], the ACLs and Kerberos authentication are used by the file systems that have powerful security, for example AFS [13, 7] and Coda [14, 13]. The use of ACLs is a good instance of possible future improvements of ZFS.

3.4   Synchronizing the Cached Volumes

When a volume is cached on a node, it is necessary to synchronize its contents with the node's volume master so that the node could see the changes made by the other nodes and vice versa, because the local version of the volume is used by file operations performed by the node. The files and directories are being synchronized in two directions. The local versions get *updated* according to the master versions and the local modifications get *reintegrated* to the volume master.

3.4.1   Modification Log

The file system needs to maintain a modification log for volumes that are cached on a local node so that it could support disconnected operation and could reintegrate the local changes effectively. Moreover, if both the local node and the volume master made changes, it would be impossible to decide what changes were made by the local node and what changes were made by the volume master without the modification log.

The log should contain all operations modifying the state of file system. The question is whether the file system should employ one log for all volumes cached on the local node, one for each cached volume or a separate log for every directory and file.

If one log was used for all volumes or one for each volume, it would be expensive to search for records for a specific file or directory when reintegrating a given file or directory. If one log was used for all volumes, it would also be inefficient to delete corresponding log entries when the node chooses not to cache the volume anymore, or the file system would simply ignore such entries. On the other hand, searching for the corresponding entries is trivial when using separate modification logs for every file or directory. Although it is slightly more difficult to manage the number of log files, it is a better solution. For example, we can easily delete useless log entries (see sections 4.6.1 and 4.6.2).

Another reason, which is another argument for the usage of separate logs for each file and directory, is that there are only few types of log entries as described below.

1. In a directory, a user can only *add* or *delete* a directory entry. It is obvious that simple operations like `create` or `unlink` can be represented by these log records. But it is possible to represent the more complicated operations too: `link` simply adds a new directory entry for the same file, `rename` deletes one directory entry and inserts another one.

2. For a file opened for writing, the only type of log record is a modification of data in an interval of offsets of the file. The data do not need to be stored to log because they are in the file.

3. The user may also change the attributes (ownership and access rights) of a file or a directory. This does not need to be written to the log, it is enough to remember the old attributes that the file had when it was last updated. The old attributes may be kept in metadata, the new attributes are in the file system.

3.4.2 Detecting the Modifications

While the file system operations are being processed, the file system checks whether the files or directories used by the operation should get synchronized. It is essential to find out whether the file has changed on the volume master or on the local node and how it has changed. Eventually, the update or reintegration is started, or the conflict is represented in the file system.

Because of the reasons described in the previous section, it is necessary to maintain a modification log for the changes that are to be reintegrated to volume master. The question is what is the best way to detect the changes made by the volume master.

The first idea is to use a modification log on the volume master to update the local copy of the volume. The obsoleted records in the log have to be deleted so that the log would not occupy the whole disk. But the log entries are not obsoleted while other nodes still need them. So the volume master would have to wait until all its descendants in the volume hierarchy update the corresponding files. But the descendants may be disconnected for an unlimited amount of time, or they may be "dead" and not deleted from the list of nodes. Therefore, the log could have an unlimited size and this is what should be avoided.

Another option is to change the file handle[1] after each modification of the file. This is not a good solution because the file identifier is used to identify a specific file. Therefore, it would look like the file was deleted and new one was created, which would cause more expensive synchronization.

The best solution is to assign each file a version number and increase it each time the file gets modified. It is simple to find out what file has changed from the version numbers. The local file has changed when its version number

---

[1]Most file systems use low-level file identifiers that are easy to work with. In distributed file systems, such identifiers are often called file handles and usually consist of an ID of the node, an ID of the file on the node and a generation number used to distinguish the recycled file ID.

is greater than the master's version number that the file had when it was last
updated. The file on the volume master has been modified when the master's
version number has changed since the file was last updated.

The version numbers are needful only for regular files and directories
because these are the only file system entities that ought to be synchronized.
Character devices, sockets and pipes do not have internal data, symlinks
have to be deleted and recreated to change their contents, and finally it is
not a good idea to synchronize the contents of block devices because the
same devices do not have to be attached to all nodes.

3.4.3   Update

As described earlier, the modifications of a file on node's volume master are
detected only by comparing the version numbers of the file. So it is necessary
to find out what exactly has changed. The method is different for regular
files and directories.

When updating a regular file, it is necessary to update only the parts of
the file that have not been updated yet and were not modified by the local
node. These parts can be updated in several ways.

First option is to simply get all the wanted parts of the file and write the
data to the local version of the file. But this solution fetches also the blocks
that were not modified by the volume master so it may cause useless network
traffic.

A better solution is to fetch only the parts that were modified by the
volume master. In order to do so, the hash sums of small blocks[2] would be
computed by the local node and by the volume master. If the hash sums
differ, the corresponding blocks differ too and have to be updated. This
solution is used by ZFS.

When updating a directory, the changes made by the volume master have
to be found and then performed in the local cache. To find the changes, the
contents of the local and the master directory have to be read and compared
while ignoring the changes made by the local node.

Regular files are being updated in the background when the local node is
connected to the volume master via a fast network, and they are accessible
during the update. If a user wants to read a block that has not been updated
yet it is fetched from the volume master. On the other hand, the update of a
directory has to finish before the file system operation can continue because
it must see the updated directory.

---

[2]The maximal size of block for which the hash sum in computed is 8 KB, which is the
maximal size of data block accepted by the `read` and `write` requests. The hash function
currently used is MD5.

When updating a directory, it is not a good idea to completely update all files and directories that the directory contains, because the update of the volume root would cause the update of the whole volume. Therefore, the files and directories that are contained in the directory being updated are created as empty, and they get updated when the user opens them or performs a directory operation in them.

3.4.4 Reintegration

When a file or directory was modified by the local node, it is necessary to reintegrate the changes to the volume master. The reintegration is easier than the update because the changes are described in the modification log (see section 3.4.1). To do so, the appropriate log entries are read, the operations are invoked on the volume master, the log entries are deleted, and the file versions are updated.

Similarly to the update, the regular files are being reintegrated in the background when the local node is connected to its volume master via a fast network, and it is possible to access them meanwhile. The directories have to be reintegrated before the file system operation can continue. The reason is that a conflict can be detected and represented in the file system during the reintegration (see section 4.4.1), and the operation should see the conflict.

When the reintegration is started, it would be possible to reintegrate everything modified or just the file or directory being accessed. Both options have several advantages and disadvantages as follows.

When all modifications are being reintegrated at once, it is good that the modifications are visible to the other nodes as soon as possible. On the other hand, the file system operation has to be delayed until at least all directories are reintegrated, which could take a long time especially when a disconnected node that made many changes connects again. It would also be very complicated to allow an execution of other file system operations during the reintegration.

It does not take such a long time to reintegrate the directory that is used by the file system operation. This type of invocation of the reintegration is also similar to the invocation of the update. This method is much better than the previous one when separate logs for each file and directory are used (see section 3.4.1, the main benefit of separate logs is their optimization). However, all modified directories and files have to be visited in order to reintegrate them, but the user can do this manually. To help the user, ZFS could provide a special tool that would reintegrate all modification logs. With respect to the previous reasons, this type of invocation is used by ZFS.

3.5 Conflicts

Conflicts appear when node `N`, which caches a volume, starts to update or reintegrate a file or a directory and another node has invoked conflicting operations since the last time node `N` had updated and reintegrated the file or directory. So the conflicts mostly appear when a disconnected node that made changes connects again and tries to perform an operation on the modified file or directory.

3.5.1 Representation of Conflicts

Coda represents a conflicting file[3] as a symlink to its file identifier, which makes the contents of the file inaccessible. According to Coda manual [15], a user needs to run the utility called `repair` and type several commands to convert this symlink to a directory and to fix the conflict. This is a rather complicated solution.

To avoid usage of a special utility, it would be necessary to represent the conflict with the conflicting versions in the file system. This would enable the user to see and access the versions, compare them like usual files and resolve conflicts by deleting the unwanted versions (see section 4.4.2).

The first idea how to represent the conflict is to create a real directory and place all versions of the conflicting file into it, the names of versions would be the names of nodes. This approach does not work for conflicts on directories. When the directory is in an attribute – attribute conflict (see section 3.5.2), the directory representing the conflict would contain the versions of the conflicting directory. All versions of the conflicting directories should have the same contents so the contents should be hard-linked between the directories. However, the conflicting directory may contain a subdirectory but directories cannot be linked in Linux and many other operating systems.

Similar problems arise when the conflict would be represented by several files whose names would be a concatenation of the original name and the name of the corresponding node. Another problem of this solution would be that such names might already exist in the directory that contains the conflicting file.

Still, representing the conflict by a directory that contains the conflicting versions is a good idea because it is obvious which files are in the conflict. The directory has to be a virtual directory because of the reasons described above. It exists only in memory structures and contains directory entries that are used to directly access the appropriate versions of the file.

---

[3]A conflicting file is the file which is in the conflict.

Another question is whether the user should see all conflicting versions of the file, or just the version on the local node and the version on the volume master of the local node.

At the first sight, it would be better to show all conflicting versions. But it would be very inefficient because the versions are not in one place as they would be if we used a real directory for the conflict. Node would need to check the version numbers of the file on all reachable nodes each time it tries to access the file, or the node would do that only for the first time and then other nodes would tell the versions of their files when they connect or modify the file. Another possibility would be to dedicate one of the nodes to be a conflict manager for the volume, the other nodes would report their version numbers to this manager when they connect or modify a file. All these solutions result in high network traffic.

Therefore, it is better to show only the local version and the version on the volume master because the node has to compare the versions only on these two nodes. Thus the conflict is created only on a node that caused it, which has another advantage of not annoying the other nodes with the conflict that the node created and should resolve.

According to final decisions, the conflicts are represented in the file system as a virtual directory whose name is the same as the name of the conflicting file and the virtual directory is located in the file's place. In this virtual directory, the local and master version of the file are located and each of them is named according to name of the node on which the version is. If the conflict is caused by deleting one version and modifying the other, the deleted version is represented by a virtual symlink to the existing version. For example, when a regular file `foo` on `node1` is in conflict with a character device `foo` on `node2`, there is a virtual directory `foo` in place of file `foo` and it contains the regular file `node1` and the character device `node2`.

3.5.2 Types of Conflicts

There are several types of conflicts as described below.

attribute – attribute conflicts appear when the local node has changed the file attributes (access rights, user ID of the owner and group ID of the owner) of a generic file[4] in a different way than the node's volume master. A file may be in an attribute – attribute conflict and another type of conflict at the same time.

---

[4]A generic file is a file of any type, i.e. a regular file, directory, symlink, character or block device, socket or pipe.

modify – modify conflicts (version conflicts) arise when the local node has modified the contents of a regular file and the volume master has modified it too.

create – create conflicts (file handle conflicts) turn up when there is a directory entry with the same name but different file handles on the local node and on the volume master. They are a result of a conflicting `create`, `mkdir`, `mknod`, `symlink`, `link` or `rename`, or of a situation when at least one node deleted a generic file and created another one with the same name[5].

modify – delete conflicts come out when the local node has modified a regular file while the volume master has deleted it.

delete – modify conflicts occur when the local node has deleted a regular file while the volume master has modified it.

3.6 Configuration Management

In order to do its job, the file system has to know certain information, for example the list of nodes and volumes, what node it should contact to access a given volume etc. This *configuration* of the file system needs to be distributed to all nodes. There are two main options how to do this.

The first option is to employ a separate configuration manager, which is requested by other nodes to send them the configuration or to change it. Coda [14] uses this option[6]. But why should there be another system providing access to some information when there already is one?

Therefore, the better choice is to use the file system itself to manage its own configuration. The configuration must be located in a fixed place so that it could be found. The possibilities are a predefined path from the root of file system to the directory containing the configuration, a predefined path from the root of a specific volume, or a separate volume. The best solution is to store all configuration files in a separate *configuration volume.*

Another question is whether the configuration volume should be accessed remotely or cached. If it was accessed remotely, each node would need to know from which node it should read the configuration. It would have to know the host name of the node. The ID or name of the node would not be enough because the node needs the configuration to convert the ID or name

---

[5]A newly created file always gets a different file handle even if it has the same file identifier because the generation field of file handle is increased.

[6]The configuration manager in Coda is called SCM according to [15].

to the host name. The host name would also have to be stored out of the file system, therefore it would be difficult to update it when the hierarchy of the configuration volume changes.

On the other hand, when the volume would be cached on the local disk, the file system could read the complete configuration without any interaction with other nodes and build the in-memory data structures. However, the cached configuration may be outdated. So the file system has to start reading the configuration once again, which causes the configuration to be updated according to the node's volume master and to refresh the in-memory data structures.

Although the disconnected node knows where in the local file system the contents of volumes cached or provided by the node are located[7], it needs to read the complete configuration to be able to work in a usual way, for example it needs to know where the volumes are mounted.

Because of the previous reasons, the final decision is that the configuration is stored in the configuration volume[8], which is cached by all nodes.

3.6.1 Updating the Configuration

When a user of the file system changes a part of the configuration, the file system should ensure that all nodes refresh their configuration. As described in the previous section, the node that is starting the file system updates its configuration during the startup. So it is needed to describe how do the running nodes update their configuration.

The obvious option is to periodically check the versions of all configuration files and reread the ones whose versions have changed. However, this is not a good solution because all nodes would send many useless messages and would notice the change of configuration after non-trivial delay. The configuration also does not change too often.

A better solution than polling is to ensure that all nodes reread the changed parts of configuration as soon as possible after the modifications happen. First, the changes have to be detected by the node that made them. This can be done by adding hooks to functions accessing the local files of the volume. Then, the other nodes have to be notified. It is enough to notify the nodes that are the direct descendants in the volume hierarchy (see section 3.1.2). When they receive the notification, they update the corresponding files and thus modify them and send the notification to their descendants.

---

[7]This information is stored in the node's private configuration.

[8]The ID of the configuration volume is 1. In the implementation, it is better to use a predefined constant VOLUME_ID_CONFIG instead.

The whole subtree of the node that changed the configuration refreshes its configuration this way. The rest of the hierarchy is notified as follows. When the node changes the configuration, it reintegrates the files to its volume master (the ancestor in the hierarchy). Thus the volume master modifies its files, notifies its descendants except the one that has reintegrated the file, and reintegrates the file to its ancestor. The result is that all nodes were notified, and have updated and reread their configuration.

3.6.2 Adding a New Node

It is possible to do most of the changes of configuration, for example to add a volume or choose to cache a volume, by editing the files on the configuration volume. But it is more complicated to add a new node because the new node does not have the configuration yet.

New nodes can be added only by the current ones so that there would be a control what nodes are able to access the file system. An existing node adds information about the new node to the configuration. Because the new nodes do not have the configuration of the file system cached on their disks, they require a way to fetch the configuration[9]. So the file system provides a possibility to specify from which node the new node should get the configuration for the first time[10]. During the startup, the file system will fetch the configuration from the given node and cache it on the local disk.

---

[9]Copying the configuration from another node would not work because of the metadata.

[10]This can be specified by passing the `--node=ID:NAME:HOST_NAME` option to the daemon when starting it.

Chapter 4

Implementation Details

This chapter describes the important or interesting parts of the implementation in more detail. Alternatives are compared where appropriate.

4.1 Architecture

4.1.1 Kernel Space or User Space?

When writing a distributed file system, one of the important decisions is whether it should be completely located in the kernel space, or whether the majority of code should be in a user space daemon and the kernel should only contain a simple layer to redirect the VFS[1] calls to the daemon and the replies back to VFS.

When the file system is completely in the kernel, it is faster because there is no additional indirection between the kernel and the file system. The file system also may use features of the kernel that can't be used from the user space and thus implement certain functions more effectively. On the other hand, programming and especially debugging in kernel space is more complicated.

Implementing the file system by the user space daemon also has several advantages and disadvantages. The daemon is easier to write and debug. It also can be easily restarted or upgraded. Porting the daemon to another UNIX-like operating system is not difficult too. Moreover, the nodes that do not want to access the file system but only provide the volumes to other nodes do not have to change their kernel. It also should not be much slower than the kernel solution because the performance overhead of the indirection

---

[1]Virtual File System (VFS) is a generic interface in the Linux kernel that enables the kernel to access any file system.

should not be too high in comparison with the network latency, so the main reason to put the whole file system to kernel is not very important. Therefore, the user space solution was chosen.

4.1.2 Communication between VFS and the Daemon

When the file system is implemented mostly in the user space, it requires a way to send file system requests from the kernel to the user level part and possibly to send commands in the other direction.

The first idea is to use the character device that is used by Coda [14] to talk with its cache manager. This is a good solution if the set of requests used by Coda is enough for the file system and all files may be copied onto the local disk. Neither of these conditions is true for ZFS because it should allow a volume to be accessed remotely if the volume is not cached, and the kernel is required to send read/write requests because of this.

There are several projects that enable to write user space file systems, for example LUFS[2]. LUFS [10] consists of a kernel module, a user space daemon, utilities and user land file systems implemented in shared libraries. The shared libraries are linked when the file system is mounted. If it were possible to run a network server in such a shared library, the main disadvantage would be the duplication of memory data structures and of network communication. The server in the shared library would be running only when the file system is mounted. So it would be necessary to run a separate daemon to process the requests from the other nodes. The other option would be to simply redirect the requests to the daemon via another communication link. However, it would be a superfluous indirection and it would not be much simpler than implementing a kernel module.

The most powerful and efficient solution is to implement a kernel module. It does not have the disadvantages of the previous approaches and makes it possible to implement downcalls[3] easily. It also is not hard to implement so it was chosen[4]. A character device[5] is used to send packets between the kernel and the daemon, the communication protocol is the same as the protocol between nodes.

---

[2]LUFS is mentioned here just as an example, other similar projects would have similar disadvantages.

[3]A downcall is a call from the daemon to the kernel.

[4]The kernel module was written by several students of Operating Systems course as their semestral project after a suggestion of my supervisor.

[5]The major number of the character device is `251` and the minor number is `0`.

4.1.3 Overview of the Architecture

Figure 4.1 shows an overview of the architecture. One node is illustrated in more detail to demonstrate the flow inside the node. Because of simplicity, the arrows represent only the ways that the requests are sent along, the replies go in the opposite directions.

When an application wants to access a file system, the system call goes though the kernel dispatch table and is passed to the VFS, which calls the corresponding file system. When the application wants to access the ZFS file system, the request is passed to the ZFS kernel code, which packs the function number and its arguments into a message and sends it through the character device to the daemon. The daemon processes the request by invoking the operation on another node[6] or by accessing the local file system. The reply to the request is sent back to the kernel through the character device and is returned to the user application. The daemon may also send a request to invalidate a dentry in the kernel dentry cache to the kernel code through the character device.
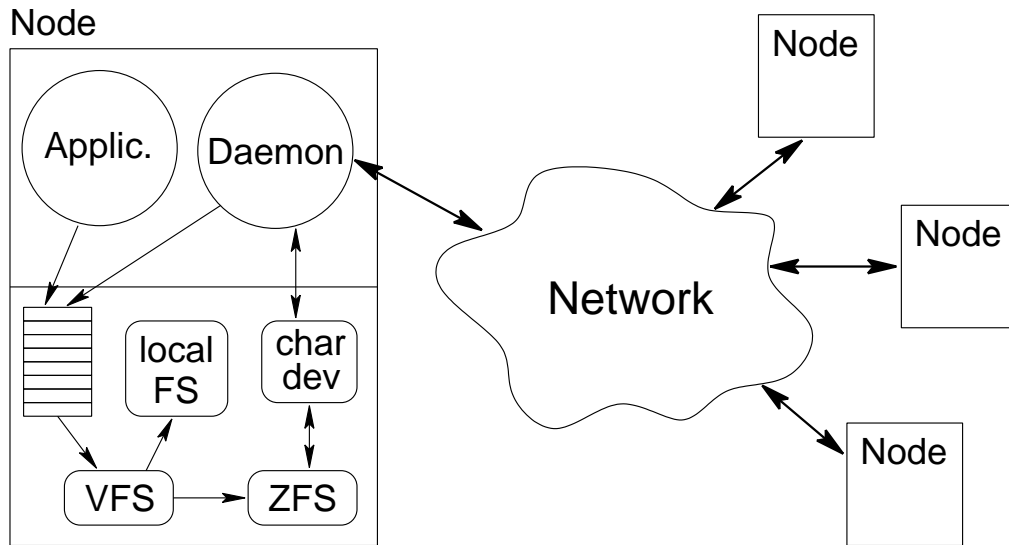

Figure 4.1: Overview of the Architecture

---

[6]The arrow to the network goes from the daemon because of simplicity. The network communication also goes through the kernel.

4.2 Communication between Nodes

4.2.1 Communication Protocol

Nodes have to invoke operations on the other nodes to be able to access the volumes that are not provided by them. *Remote procedure call* protocols are well suited for this purpose.

SUN RPC is a protocol that is well supported on many Unix platforms. According to `man` page of `rpcgen`[7] distributed together with GLIBC, it does not support multi-threaded applications and thus it could not be used. After implementing a proprietary communication protocol, I have found out that multi-threaded applications are supported by `rpcgen`. However, I did not choose to use SUN RPC because it copies all data when decoding the packets, does not support one-way requests and it would be hard to dispatch requests to several threads.

CORBA is another middleware well supported on Unix platforms. It has many good features but it is quite heavy-weight. The file system also requires a communication protocol between the kernel and the daemon but CORBA is too complicated for the kernel. It also is not a good idea to have two communication protocols so CORBA was not chosen.

Because of the previous reasons, a proprietary communication protocol is used by ZFS. It sends packets over TCP streams so it does not have to worry about network errors. The packets use the little endian encoding, the elemental data types are aligned to their size, and strings are encoded as zero-terminated strings together with their length. The daemon performs several optimizations with respect to decoding and encoding of the packets. No memory is allocated and no bytes are copied when strings or data buffers are decoded, just a pointer is returned from the decoding functions. Similarly, the `read` operation reads the data directly to the packet.

4.2.2 Connecting and Authentication

When the daemon on one node wants to send a request to another node, it checks whether there already is an established connection between these two nodes and if so, the existing connection is used. It they are not connected and the last attempt to connect was at least a predefined amount of time ago the daemon tries to connect.

After the connection is established, both nodes have to know which node is connected to the other end. Although the node that initiated the connection already knows which node it wanted to connect, it is good to verify whether

---

[7]The RPC protocol compiler.

it has really connected to the expected node to detect bugs in configuration. But the node that accepted the connection cannot find out which node has connected because nodes are allowed to connect from arbitrary address (this is a case of a laptop connected via a modem). So each node sends its name to the other node in the authentication messages.

The daemons should use a reciprocal security check to verify whether known nodes are going to be connected, but it is not implemented yet. This could be done by authenticating the nodes by Kerberos or by digitally signing the random bytes generated by the other node.

4.2.3 Measuring the Connection Speed

In order to support mobile operation, the daemon has to estimate the speed of connection so that it could limit the network traffic when the node is connected via a slow link. Because a node connected to a fast network may communicate with a node connected via a slow link, the speed of the link via which the node is connected to network cannot be used, so the daemon has to measure the speed of network between the two nodes. Latency is measured because it is probably a better parameter of the connection than throughput.

The first idea is to estimate the connection speed while the requests are being sent to the other node. The only requests that are being sent in a relatively large number are the operations that induce the target node to access its disk or send further requests to another node. These actions take a relatively long time and would distort the delay between the request and reply so they are not suitable for measuring the speed.

So it is necessary to use an auxiliary request, which is processed by the receiver as quickly as possible. The results are probably the most exact when the ICMP echo and echo reply messages[8] are used. However, firewalls may be configured to drop ICMP traffic thus making this option not well functional. Anyway, the TCP connection between the nodes can be used for sending so called ping messages. This works well although the overhead of processing the message is higher than the delay of processing the ICMP message.

The speed could be measured in regular time intervals or only when the connection is being established. The daemon expects that the connection speed does not change while it is established, and that the node has to disconnect first to change the quality of connection. This can be expected because the change of speed should happen only when the node moves in space, for instance when a laptop is taken away and connected later via a modem. So the speed is measured only when making the connection.

---

[8]These messages are used by the `ping` program.

4.3 File Handles and Capabilities

Each file system that allows a file to have several hard links has to provide low-level file identifiers. The low-level file identifiers are good for other file systems too because it is better to identify files internally by numbers and not by strings. In Unix file systems, the low-level identifier is usually called an inode.

The distributed file systems that support a remote `open` have to identify the open files too so that clients would not be allowed to read or write the files that they did not open.

4.3.1 File Handles

File handles are the unique identifiers of files. They consist of the ID of the node that created the file handle, the ID of the volume on which the file is, the device and inode number of the local file in which the file is stored and finally the generation number, see figure 4.2. The generation is increased each time the file in the local file system is deleted and thus a newly created file with the same first four numbers differs in the generation.

Node ID	Volume ID	Device	Inode	Generation

Figure 4.2: ZFS File Handle

The operations performed on the local file system require the local paths to the file handles to be determined. Although ZFS provides a possibility to build the local path to the file handle from the information in the metadata[9], it is important to keep the path information in memory because it is much faster to create the path from the in-memory data structures than from the metadata.

One of the options is to cache whole paths for file handles. But it would be difficult and inefficient to update such a cache. For example, when a directory is renamed or moved, it would be necessary to update or invalidate the cached paths for its whole subtree. It would be good to have a data structure that could handle such situations.

Fortunately, there is a solution that does not have the problems of the previous option. The main idea is to remember the components of the path separately and to concatenate the path when it is required. Because the file handle may have several hard links, it is necessary to separate the name from

---

[9]The local path can be built using the lists of hard links of the file handles.

the in-memory representation of the file handle. The in-memory structure holding the name of the file, a pointer to the parent, and a pointer to the file handle is called a `dentry`. The in-memory file handle contains the list of subdentries because it must be shared between all hard links of the file handle when it is possible to make several hard links of a directory. Moreover, the file handle is used for synchronization and metadata management. These data structures solve the problem of renaming or moving a directory by changing the name in the dentry and redirecting several pointers.

Figure 4.3 shows an example of the connections between data structures that represent the directory tree. There are two directories (`dir1` and `dir2`) in the root of the volume and both of them contain a hard link (`name1` and `name2`) of the same file handle (`FH4`). The second directory contains another file whose name is `file`.
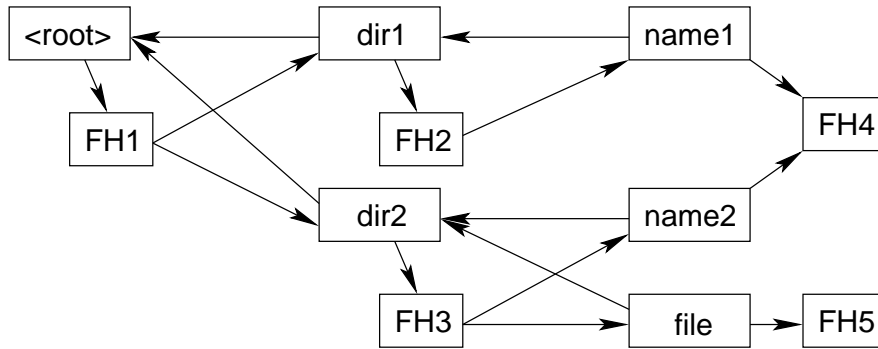

Figure 4.3: In-Memory Representation of Directory Tree

The file system must assign the file handles for special file types too. The special files are the virtual directories (the components of paths to volume roots), the conflict directories and the non-existing files (the deleted files in the delete – modify or modify – delete conflicts). ZFS exploits the fact that certain numbers cannot be used as IDs of the nodes and volumes so the various combinations are assigned to the virtual files. The node ID must not be a NODE_NONE which is defined to 0 and the volume ID cannot be a VOLUME_ID_VIRTUAL which is also defined to 0. The assignment of node and volume IDs is outlined in the following table.

Node ID	Volume ID	Type of the file
$= 0$	$= 0$	Virtual directory
$= 0$	$\neq 0$	Conflict directory
$\neq 0$	$= 0$	Non-existing file
$\neq 0$	$\neq 0$	Real file

4.3.2 Capabilities

As mentioned earlier, it is necessary to identify open files. They are identified by *capabilities* in ZFS, although ZFS capabilities are slightly different from the usual capabilities, which consist of a file identifier, access rights and verification [19, 11, 17]. ZFS capabilities consist of the file handle, the flags the file was opened with and the verification, see figure 4.4.

File handle	Flags	Verification

Figure 4.4: ZFS Capability

Each node should be able to verify the capabilities that are sent to the node within the requests. So that it could be possible, the capability must be generated by the node. When the node is accessing the volume remotely, it has to know the capability generated by the volume master too, and the capability received in the request must be replaced by the master capability before invoking the operation on the volume master. So a mapping from the local capability to the master one must be remembered (in memory).

Because the daemon keeps the local and master capability in memory, the verification does not have to be encrypted. It suffices to compute the verification as a hash sum from the rest of the capability and from several random bytes[10], and compare the verification of the received capability and of the capability kept in memory. When the file is closed, the capability is destroyed.

4.4 Creating and Resolving the Conflicts

4.4.1 Creating the Conflicts

When a conflict is discovered, the conflict directory is created and the two (local and master) conflicting versions are placed into it. Different types of conflicts (see section 3.5.2) are detected during different actions.

attribute – attribute and modify – modify conflicts are found while updating the conflicting file or the directory containing the file.

create – create conflicts are detected during the update of the directory containing the conflicting file or during the reintegration of the `add` log entry.

---

[10]It would be enough to use the random bytes themselves as the verification but it is not a good idea to show the output of the random generator to the user.

modify – delete conflicts are found when updating the directory in which the file is located.

delete – modify conflicts are discovered when the corresponding `del` log entry is being reintegrated.

4.4.2 Resolving the Conflicts

Because a conflict is represented in the file system as a directory containing the versions, it is easy to resolve the conflict. To do so, the user can simply "delete"[11] one version and thus select the other to be the correct one. The daemon handles the deleting in the conflict directory in a special way.

The actions are different for different types of conflicts:

attribute – attribute conflicts

The daemon simply sets the attributes of the version being "deleted" according to attributes of the other version.

This type of conflict may also be resolved by changing the attributes so that they do not conflict anymore.

modify – modify conflicts

When the user "deletes" the local version of the file, the local changes have to be lost and the file updated according to the master version. So the daemon deletes the list of modified intervals[12] of the file and the list of updated intervals[13]. Finally, it is necessary to change the versions so that the file would look as it was not modified by the local node and was modified by the volume master, and schedule the file for updating.
Another option would be to simply delete the file and let it completely update again but it would result in higher network traffic.

When the user wants to "delete" the master version, the list of updated intervals[13] is added to the list of modified intervals[12], and the versions are changed to mean that the file was not modified on the volume master. Finally, the file is scheduled for reintegration.
The method above is a need, not an option. The master version of the file cannot be simply deleted because the local file does not have to be completely updated, so corresponding data would be lost if the master version was deleted.

---

[11] `unlink` a non-directory or `rmdir` a directory, which does not have to be empty.
[12] A list of intervals that were modified by the node.
[13] A list of intervals that were updated according to the volume master.

create – create conflicts

>   When resolving this type, the corresponding file is really deleted, or moved aside[14] if the file with corresponding file handle exists on the other node. This makes a place for the other version and the file is created in this place when the directory containing the file is being updated or reintegrated.

modify – delete and delete – modify conflicts

>   When the user confirms the deletion of the modified file by "deleting" it, the daemon really deletes the file.

>   When the user discards the deletion of the modified file by "deleting" the virtual symlink representing the deleted file, the modification log for the directory containing the file is modified to make the daemon update or reintegrate the file next time it verifies the versions of the directory. An `add` log entry is added when resolving modify – delete conflict to force the file to be reintegrated, or the log entry that caused the delete – modify conflict to be created is deleted.

4.5   Invalidating the Kernel Dentries

When the kernel wants to lookup a file, it tries to find it in the dentry cache. If the dentry is found there and is still valid, it is used. Otherwise the kernel invokes a `lookup` operation and creates a new dentry.

So when the daemon knows that a directory entry was deleted or moved, it is necessary to invalidate the dentry in the cache. Although the kernel would invalidate the dentry automatically after some time, it is necessary to invalidate it as soon as possible because the file should disappear immediately.

When the daemon deletes, renames or moves a file, creates or resolves a conflict, its sends a one-way invalidation request to the kernel so that the view of the file system presented to a user would be up-to-date.

4.6   Metadata

The ZFS file system requires more metadata than the local file system. The additional information is stored in the directory `.zfs` in the root of the volume in the local file system, this directory is invisible via ZFS.

Using an auxiliary file for each file of the file system would take too much space because a whole block is usually allocated for small files. This can be

---

[14]The files are moved to the shadow directory in some situations, see section 4.7.

easily avoided for fixed length metadata like file handle, local version, master version etc. These fixed length metadata are stored in a hashed file, which automatically grows and shrinks. There is one more hashed file that contains the mapping of master file handle to the local one.

Unfortunately, all types of metadata do not have a fixed length. Such metadata are stored in separate files whose names are generated from the file handles and the types of metadata. To avoid the efficiency limits of directories[15] of the local file system, these files are stored in an appropriate directory of the directory tree whose components also have names generated from the file handles.

The first type of variable-length metadata is the list of hard links of the file. Each hard link is represented by its file name and a part of the file handle of the directory containing the file. However, if the file has only one hard link and its name is short enough, which is actually the most common case, the information about hard link is stored in the hashed file and not in the additional file.

The various modification logs, which describe the changes made by the local node and are used during the reintegration and update, are stored in separate files too because of the reasons stated in section 3.4.1. The types of modification logs are described in the following sections because they are important and interesting optimizations are performed on them.

An example of locations, as seen from the volume root, of various types of metadata is in figure 4.5.

```
.zfs/metadata
.zfs/fh_mapping
.zfs/3/5/000003030013C053.modified
.zfs/3/7/000003030013C073.updated
.zfs/5/E/000003030013C0E5.hardlinks
.zfs/D/A/000003030013C0AD00000001.journal
```

Figure 4.5: Example of Metadata Locations.

4.6.1 Journals for Directories

There is a separate journal for each directory that contains the changes that have not been reintegrated yet. As mentioned in section 3.4.1, just two types

---

[15]Many local file systems store the directory entries in a list and the lookup of a specific name takes $O(n)$ time because of that. So it is better to prevent from storing too many files in one directory.

of journal records are necessary, the `add` record and `del` record that mean that a directory entry was added or deleted, respectively.

Each record contains the name and the file handle of the added or deleted directory entry. It also contains the master file handle[16] so that the file system would know what file it should link, move or delete on the volume master or could detect create – create conflicts, and also the master version[17] so that it could detect delete – modify conflicts.

The journal can be optimized for size. When there is an `add` record and later a `del` record with the same name in the journal, such a pair of records may be deleted without loss of information. There remain only records for files that originally were in the directory and were deleted, and for files that did not exist and were added to the directory. So the space complexity of the optimized journal is $O(n_1 + n_2)$ where n_1 is the original number of files after last reintegration and n_2 is the current number of files in the directory.

When the daemon creates an in-memory file handle for the directory, it reads the corresponding journal into a special data structure[18]. The data structure keeps the journal optimized when entries are being added to it. The optimized journal is written back to the file after it is read to save the disk space. When the daemon wants to add a new entry to the journal, it appends the record to the journal file and inserts it to the data structure. During the reintegration, the reintegrated journal entries get deleted so the modified optimized journal has to be written to the file again.

4.6.2 Interval Files

For each regular file, there is a file containing the list of modified intervals of the file (i.e. the modification log) and a file with the list of intervals that were updated according to the volume master in the metadata directory tree. If the file was not modified or it was completely updated, the corresponding interval file does not exist to save the disk space.

The corresponding interval files are read when the regular file is being opened. The intervals are added to an interval tree that merges the adjacent intervals and unions the overlapping ones and thus consolidates the interval list. The consolidated interval list is written to the file after the complete original list has been read to save disk space. When a new interval is to be added, it is appended to the interval file and inserted to the interval tree.

---

[16] A file handle of the file as used by the volume master of the local node.

[17] The version of file that was downloaded from the volume master.

[18] A hashed doubly linked list is used because the records should be in the order in which they were added and should be looked up in $O(1)$ time because of the journal optimization.

When the daemon is closing the regular file, the optimized interval lists are written to interval files again.

4.7 Shadow Directory

When a file or directory is being moved from one directory to another one, a `del` journal entry is added to the source directory and an `add` entry to the destination directory. If the `del` entry is being reintegrated first it is necessary to make the directory entry available but keep the file or directory aside until the `add` entry gets reintegrated, which will move the file or directory to its new place. The place where the file or directory is located meanwhile is called a *shadow directory*.

The files or directories are being moved to the shadow directory in one more situation. When a node is updating a parent directory of directory D and directory `D` was deleted by the volume master, the directory `D` is deleted during the update. However, the node might have created or modified files or create directories in the subtree of directory `D` but such files or directories should not be deleted. It would be inefficient to check the whole contents of directory `D`, and create a conflict in this situation, so the new and changed files and the new directories are moved to the shadow directory. The user has to manually check whether these files really should be deleted.

The shadow directory is a directory `.shadow` in the root of each volume. The shadow directory is visible via ZFS only to the local node so that the user could access its contents and move the files or directories to an appropriate place. The files and directories are stored in a directory tree similar to the directory tree for variable-length metadata and their names consist of the original name and the file handle to avoid name conflicts.

4.8 Configuration

The configuration consists of the global configuration, which is shared by all nodes and is stored in the configuration volume (see section 3.6), and of the local configuration, which is private to each node.

4.8.1 Local Configuration

The local configuration contains information that is specific to a given node and should not be accessible by other nodes. It consists of few files.

config

This is the main local configuration file, which is usually located in `/etc/zfs/config`. If it has a different location it must be specified in the `--config=FILE` option of the daemon.

The file contains several configuration directives on separate lines. The valid line consists of a name of the configuration directive and its value, which is separated from the name by at least one white space character. Everything after a hash ('#') is ignored. If the value contains spaces or hashes it must be enclosed in quotes or double quotes. The supported configuration directives are listed below.

LocalConfig – the path to the rest of local configuration files, the default is `/etc/zfs`.

KernelDevice – the name of the character device for communication with the kernel, the default is `/dev/zfs`.

MetadataTreeDepth – the depth of the directory tree containing the files with variable-length metadata, the default is `1`.

DefaultUser – the name of the local user. When a mapping for the file system UID is not defined, a mapping to this user is used. The default is `nobody`.

DefaultUID – the ID of the local user. This directive is an alternative to the DefaultUser, the default is the UID of `nobody`.

DefaultGroup – the name of the local group. This directive is similar to the DefaultUser, the default is `nogroup` or `nobody`.

DefaultGID – the ID of the local group. This is an alternative to the DefaultGroup, the default is the GID of `nogroup` or `nobody`.

this_node

There is just one line in this file and the line holds the ID and the name of the local node in the format `ID:NAME`. The local node needs this information to connect to another node to read the global configuration.

volume_info

This file holds the information about volumes provided or cached by the local node. There is one line for each such a volume. The line contains the ID of the volume, the path to the root of the volume in the local file system, and finally the limit of the size of the cached volume[19] in the format `ID:PATH:LIMIT`.

---

[19]The possibility to limit the size of the cached volume is not implemented. The value of `0` means that the size is not limited.

4.8.2 Global Configuration

The global configuration contains the information that is shared by all nodes. It consists of several files stored in the configuration volume, which is cached by all nodes and is automatically updated (see section 3.6.1) when it changes. All files of the global configuration contain one record per line and the fields are separated by a colon (':') unless something else is mentioned.

node_list
> Nodes are listed in this file. There is the ID of one node, its name and host name on each line.

volume_list
> There is a list of volumes in this file, each line consists of the ID of one volume, the name and the mount point of the volume.

volume/
> This directory contains many files and the files have the same names as the volumes. Each file describes the hierarchy of the corresponding volume. There is one node name indented by several spaces on every line. The node providing the volume is on the first line and is not indented at all. The volume master of a given node is the first node above the given node that is indented by fewer spaces than the given node. For example, the volume hierarchy from figure 3.2 on page 22 would be described as follows.

```
N1
 N2
 N3
   N4
   N5
   N6
 N7
   N8
   N9
```

user_list
> The list of users of the file system is located in this file. Each line contains the ID and the name of one user.

group_list
> The structure of this file is the same as the `user_list` file but it contains the list of groups instead of the list of users.

user/

The files in this directory describe the mapping between the file system
user IDs and the node user IDs. The name of almost every file is the
name of the node for which the mapping is. The only exception is the
file whose name is `default`, which holds the default user ID mapping,
i.e. if a mapping for a given user is not found in the mapping for
the corresponding node the default mapping is used. Each line of any
mapping file consists of the ZFS user name and the node user name.

group/

This directory is almost the same as the `user` directory, the difference
is that the files in this directory describe the mapping of groups instead
of users.

Chapter 5

Conclusion

In this thesis, the distributed file system that fulfills all proposed goals was designed and the implementation shows that it is viable.

One of the main contributions is that the file system allows certain volumes to be accessed remotely without storing any information on the local node, and other volumes to be transparently cached in the local file system at the same time. When accessing the cached volume, the cache is used by the file system operations and is automatically synchronized while the node is connected via a fast link. When connected via a slow link, the whole files are not synchronized to limit network traffic but the blocks that were requested by the `read` operation and were not updated yet get updated. When updating a file, only the parts that differ are updated in contrast to Coda and the file is accessible during its synchronization.

Because the Coda client requires all volumes it wants to access to be cached in the cache file or device, the files of the volume provided by the node have to be copied to this cache too. This is a useless duplication of data. On the other hand, ZFS works with the volumes that are provided by the local node directly via invoking the operations on the files or directories of the volume in the local file system, as it does for the cached volumes. The data are not copied and the only additional information is the metadata.

The representation and resolution of the conflicts is superior to Coda. In contrast to Coda, which uses a special utility to convert the conflicts to directories and to fix the conflicts, ZFS represents the conflicts in the file system as virtual directories and thus it enables to resolve the conflicts by deleting the unwanted versions.

The configuration management in ZFS is also much better than in Coda. Contrary to Coda, which uses a separate configuration manager, ZFS stores the configuration in the file system and automatically updates the changed configuration files to all reachable nodes.

Bibliography

[1] P. J. Braam. *File Systems for Clusters from a Protocol Perspective.* Second Extreme Linux Topics Workshop, 1999.

[2] P. J. Braam et al. *InterMezzo.* http://www.inter-mezzo.org, 2003.

[3] P. J. Braam, P. A. Nelson. *Removing Bottlenecks in Distributed Filesystems: Coda InterMezzo as examples.* Proceedings of Linux Expo, 1999.

[4] S. Ghemawat, H. Gobioff, S.-T. Leung. *The Google File System.* Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, 29–43, 2003.

[5] B. von Hagen. *Using the InterMezzo Distributed Filesystem.* http://www.linuxplanet.com/linuxplanet/reports/4368/1, 2002.

[6] J. H. Hartman, J. K. Ousterhout. *Zebra: A Striped Network File System.* Proceedings of the Usenix File System Workshop, 71–78, 1992.

[7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, M. J. West. *Scale and Performance in a Distributed File System.* ACM Transactions on Computer Systems, Volume 6, No. 1, 51–81, 1988.

[8] J. J. Kistler, M. Satyanarayanan. *Disconnected Operation in the Coda File System.* ACM Transactions on Computer Systems, Volume 10, No. 1, 3–25, 1992.

[9] E. K. Lee, C. A. Thekkath. *Petal: Distributed Virtual Disks.* Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, 84–92, 1996.

[10] F. Malita. *LUFS.* http://lufs.sourceforge.net, 2003.

[11] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, H. van Staveren. *Amoeba: A Distributed Operating System for the 1990s.* IEEE Computer, Volume 23, Issue 5, 44–53, 1990.

[12] G. J. Popek, B. J. Walker. *The LOCUS Distributed System Architecture.* MIT Press, 1985.

[13] M. Satyanarayanan. *A Survey of Distributed File Systems.* Annual Review of Computer Science, Volume 4, 1989.

[14] M. Satyanarayanan et al. *Coda.* http://www.coda.cs.cmu.edu, 2004.

[15] M. Satyanarayanan, M. R. Ebling, J. Raiff, P. J. Braam, J. Harkes. *Coda File System User and System Administrators Manual.* http://www.coda.cs.cmu.edu/doc/ps/manual.ps.gz, 2000.

[16] S. R. Soltis, T. M. Ruwart, M. T. O'Keefe. *The Global File System.* Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems and Technologies, 319–342, 1996.

[17] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp. *Beyond UNIX – A True Distributed System for the 1990s.* Proceedings of UKUUG Summer Conference, 251–260, 1990.

[18] C. A. Thekkath, T. Mann, E. K. Lee. *Frangipani: A Scalable Distributed File System.* Proceedings of Symposium on Operating Systems Principles, 224–237, 1997.

[19] F. Zavoral. *Distribuované operační systémy.* The textbook for the course of the same name, Charles University, 2003.

Appendix A

Enclosed CD

The enclosed CD contains the following data:

doc/	programming documentation in the HTML format generated by Doxygen from the sources of the user space daemon
thesis/	LaTeX sources of this thesis
thesis.ps	this thesis in PostScript format
thesis.pdf	this thesis in PDF format
zfs.patch	patch with ZFS File System support for Linux 2.6.x kernels[1]
zfsd/	sources of the user space daemon[1]

---

[1]The latest version can be found on my home page: `http://zlomek.matfyz.cz`