Westfälische Wilhelms-Universität Münster

Seminar thesis

Distributed file systems

for the seminar Parallel Programming and Parallel Algorithms

Thomas Hollstegge

Professor: Prof. Dr. Herbert Kuchen
Supervisor: Dipl.-Wirt.Inform. Christian Hermanns
Department of Information Systems
Practical Computer Science

Contents

1 Introduction **1**

2 Basics **1**

2.1 Fundamentals of storage 1

2.2 Characteristics and requirements of distributed file systems 3

2.3 Abstract file service model 4

2.3.1 General description 4

2.3.2 Operations 5

2.3.3 Further issues 6

3 Distributed file systems - Case studies **7**

3.1 Network File System (NFS) 7

3.1.1 History 7

3.1.2 General description 8

3.1.3 Classification 10

3.2 Andrew File System (AFS) 13

3.2.1 History 13

3.2.2 General description 14

3.2.3 Classification 17

3.3 Other approaches - the Lustre filesystem 19

4 Summary and Outlook **20**

1 Introduction

With the advent of electronic communication networks the electronic exchange of information has become a vital part of many computing systems. These systems enable people to work spatially and timely distributed. This possibility has lead to new business models in the past like the provision of 24/7 customer services for customers from all around the world or the analysis of financial information from stock exchanges in different countries. Access to a common data collection is often crucial to conduct this kind work. To enable this distributed access to a common data collection, different file systems that allow distributed access have been developed in the past. Given the economical relevance of the topic, a structured discussion about underlying principles, file systems used in practice and how they meet these principles is indispensable.

For this purpose this elaboration will first introduce foundations of distributed file systems in section 2. The basic concept of storage and requirements for distributed file systems are identified in sections 2.1 and 2.2. Section 2.3 introduces an abstract model that meets these requirements. A characterisation of widely-spread distributed file systems follows in section 3, namely the *Network File System* (NFS) in section 3.1 and the *Andrew File System* (AFS) in section 3.2. For each of these examples a history outline will be given before the systems will be described in detail. Each system will also be classified according to the abstract model introduced in section 2.3. This helps to identify the requirements that are met by these systems. After describing NFS and AFS in detail, section 3.3 will give a short overview of another distributed file system approach, namely the *Lustre* file system.

2 Basics

This section will introduce fundamental basics of distributed file systems, beginning with an introduction to storage in general (see section 2.1). After the identification of basic requirements for distributed file systems (see section 2.2) an abstract file service model that can be seen as a basis for distributed file systems will be presented (see section 2.3).

2.1 Fundamentals of storage

In computer science, the storage of data is an abstract concept that has three main characteristics. Data is encapsulated in objects that can be referred to by names.

These objects are explicitly created and deleted. Furthermore, they are not affected by system failures [Sat89, p. 150]. This storage of data allows applications to maintin data even when processes are terminated [Tan03, p. 407]. A refinement of this abstract model is a *file system*: Data is stored in *files* that are persisted on an underlying storage device and can be referred to by file names. The management of these files is realized by a part of the operating system, the file system [Tan03, p. 407].

Operating systems themselves can act in different environments with different requirements. [Sat89] has developed a model with four levels of abstraction to deduce requirements for file systems in these environments. The first level of abstraction addresses issues for a single user at a single site, using a single-thread operating system (OS). In this case facts like the naming of files and the access to these files becomes important. This level also cares about the mapping of files onto physical storage as well as integrity aspects [Sat89, pp. 150–151].

In the second abstraction level a user is considered to use a multi-thread OS at a single site. In this case concurrency control becomes an important point, as multiple processes may share the same resource, namely a file. Concurrent access may be achieved by implementing serialisability of transactions. Parallel execution of programs that are accessing a single resource may also lead to deadlocks; the detection and avoidance of these deadlocks has to be concerned on this level as well [Sat89, p. 151].

The third abstraction level describes a setting where a system is used by more than one user at a time. In this case a storage system has to incorporate means to create security. Users have to be authenticated and authorised to use resources like storage. The aggregation of users into groups is of interest as well as the issuance and revocation of access privileges [Sat89, p. 151].

While the three levels presented so far assume that resources are available at a single site, the last level covers aspects for distributed systems. These spatially distributed systems share storage resources over a communication network. In this context efficiency becomes important: Applications should continue to perform satisfactory when accessing a remote storage resource. Besides these new aspects other already mentioned points like user authentication, concurrency control and atomicity have to be revised [Sat89, p. 151].

2.2 Characteristics and requirements of distributed file systems

The last section already identified some basic requirements of distributed file systems by comparing different use cases. This identification will now be refined by compiling a structured overview of general requirements for distributed file systems. These requirements were first examined in [Sat89] where the author lists the demands he tried to fulfil with a newly developed distributed file system [Sat89, pp. 158–160]. The provided list has been widely accepted and was predominant in the literature. With a few enhancements this list can still be found in the recent literature [CDK01, pp. 314–316]. As most of these demands can be derived from requirements for distributed systems in general, the following list of requirements will only focus on DFS-specific aspects.

Transparency In general, a system provides transparency if the user is unaware of the internal separation of components inside the system [CDK01, p. 23]. In the context of distributed file systems several aspects of transparency become important. *Access transparency* enables users to access remote files in the same way they access local files, i.e. by using the same operations [CDK01, p. 315]. *Performance transparency* allows user processes to behave reliably under different load circumstances. In a distributed file system file names should be independent from the real location of the file (*location transparency*). Additionally a distributed file system should be expandable without changes to the system itself (*scaling transparency*).

Availability The system should be fault tolerant, i.e. a failure in either a client or server system should not affect the overall system functionality. Single points of failure should be avoided to achieve availability, e.g. by using redundancy or failover services for essential systems [Sat89, p. 160].

Concurrent updates If different users and/or processes access the same file at the same time, concurrency issues arise. A distributed file system should provide means to allow simultaneous access to a file while avoiding interference between processes, e.g. by using file locks (semaphores) [CDK01, p. 315].

Replication Keeping multiple copies of a file on different members of a distributed file system helps to share the load between servers. This leads to a better scalability as well as a better availability. If a server holding a file copy is not

available at some point, clients may simply request that file from a different server.

Hardware and software heterogeneity A typical computer network consists of different types of computers with varying hardware and software equipment. A distributed file system should be implemented with respect to this plethora of different systems. A common approach is to develop a protocol in a platform-independent language and implement this protocol in platform-specific software [CDK01, p. 16].

Consistency A DFS should provide a mechanism to ensure the consistency of a file's contents if multiple users and processes access this file. This may be realized by a propagation of file changes to all clients currently accessing the same file. This point is very similar to replication and transparency, but has some pitfalls of its own. E.g. network delay may lead to violation of the file contents' consistency.

Security Security aspects can be divided into three broad categories. *Confidentiality* is assured if an information is only accessible by authorised parties, *integrity* describes the protection of information against unauthorised changes, and *availability* describes the need that users can access their data whenever needed [CDK01, p. 18].

Efficiency The performance of a distributed file system should be as good as as a usual file system's performance, at least from a user perspective. Users should not have to decide whether their files should be stored on a remote or a local storage facility [CDK01, p. 316] [Sat89, p. 160].

2.3 Abstract file service model

2.3.1 General description

As mentioned before the different requirements for DFSs stated in section 2.2 have been widely discussed in the literature. To allow a structured discussion and classification of different DFSs, an abstract file service model meeting most of the identified demands was developed in [CDK01]. The model consists of three parts which are depicted in Fig. 1.

The first part is realized by a *flat file service* on the server. This service stores the contents of a file on the server's hard disk. Each file can be referred to by a unique file identifier (*UFID*). This unique identifier is created once a file is created. The

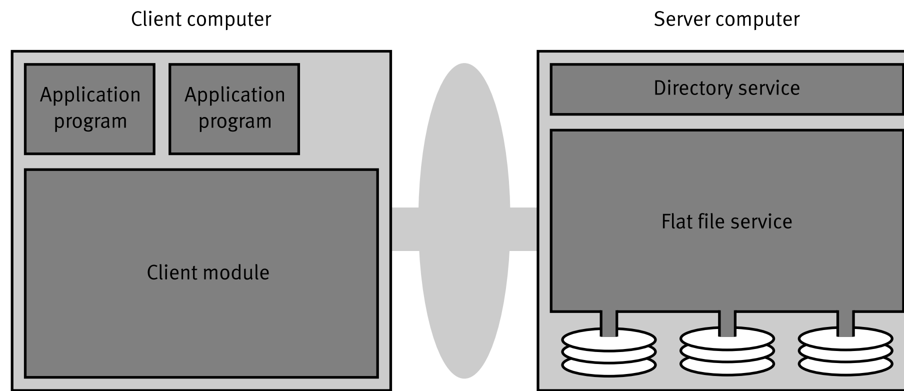

Figure 1: Abstract file service model (Source: [CDK01, p. 318])

second part consist of a *directory service* which maps textual file names to UFIDs. This service is able to create and remove directories as well as adding and removing files from a directory. Directories are stored as regular files in the flat file service and can therefore be referred to by a UFID. The third part of the abstract model is a *client module* residing in the client computer. This module encapsulates access to the directory and flat file service and provides a mapping of local file system calls to remote operations. Additionally the client module is responsible for client-side caching of files [CDK01, pp. 318–319].

2.3.2 Operations

Besides the general model [CDK01] also provides a list of reference operations for the flat file and directory service. The list can be seen in table 1.

Service	**Operations**
Directory service	*Lookup(Dir, Name)* → *FileId* – throws *NotFound* *AddName(Dir, Name, File)* – throws *NameDuplicate* *UnName(Dir, Name)* – throws *NotFound* *GetNames(Dir, Pattern)* → *NameSeq*
Flat file service	*Read(FileId, i, n)* → *Data* – throws *BadPosition* *Write(FileId, i, Data)* – throws *BadPosition* *Create()* → *FileId* *Delete(FileId)* *GetAttributes(FileId)* → *Attr* *SetAttributes(FileId, Attr)*

Table 1: Directory and flat file service operations (Source: [CDK01, pp. 319–322])

The directory service operations are called by the client module via *Remote Procedure Call* (RPC). The main functionality is realized by the *Lookup* operation. This operation searches for a file with the given name in a specific directory and returns the file's UFID. If no file with the given name exists it throws an exception. *AddName* adds a new record to a directory or throws an exception if the given file name is already in use, while *UnName* removes a directory entry, throwing an exception if the given file name was not found. The *GetNames* operation is used to search for files within a directory. It accepts a regular expression as an argument and returns a sequence of matching file names within the directory [CDK01, pp. 321–322].

Similar to the directory service operations the operations exposed by the flat file system service are accessible via RPC. The operations can be divided into three parts: *Read* and *Write* are intended to handle a file's contents. They take a UFID and an index as arguments and return the next n items from the file (*Read*) or write a list of items to the file (*Write*). Exceptions are thrown if the given position is invalid. The second set of operations is engaged with the existence of files. While the *Create* operation creates an empty file and returns its UFID, the *Delete* operation deletes the file with the given UFID. Attribute operations are contained in the last operation set. *GetAttributes* returns a file's attributes while *SetAttributes* sets the attributes of a file [CDK01, pp. 319–320].

2.3.3 Further issues

The distributed setting in which DFSs are engaged leads to some challenges in the area of access control. While the access rights to a file can directly be checked against a user's id on a single computer system, a distributed system has to cope with server-side user authorisation as user information is only stored on client computers. Two different methods for user authorisation in this setting can be distinguished: Access rights are either checked upon a directory lookup (where file names are converted to UFIDs) or upon every single request. The model stated in [CDK01] does not explicitly determine which method should be used. Instead the user information is seen as implicit information that can be accessed by every operation.

The abstract model provides support for a hierarchical file structure. It allows to store references to files that contain further directories within a directory. The presented structure creates a tree with directories as inner nodes and files as leaves, starting from a root directory. The distinction between directory and file entries is done by using file attributes. Support for a hierarchical file system can be completely

implemented within the client module as the system only uses operations exposed by the directory and flat file service.

Besides a hierarchical structuring the model also allows to build sets of files, called *file groups*. These file groups may be moved between different servers, but files cannot be moved between different file groups. A file group is identified by a *file group identifier* (FGID) that is a part of all contained files' UFIDs. As FGIDs have to be globally unique, [CDK01] proposes to use the server's IP address and the current date to generate an FGID during the creation of a file group. This also allows client computers to directly determine the server they have to send requests to by deducting the IP address from a file's FGID. This lookup is only effective as long as file groups are not moved between servers. To enable relocation of file groups between servers the file service should be used to store a mapping between FGIDs and servers inside a file [CDK01, p. 323].

3 Distributed file systems - Case studies

This section describes distributed file systems that are commonly used in practice. Section 3.1 provides an introduction to the *Network File System* (NFS), while section 3.2 presents the *Andrew File System* (AFS). Both systems will be described in detail and classified according to the abstract model introduced in section 2.3. Afterwards section 3.3 gives a short overview of *Lustre*, a cluster file system that follows other principles than NFS and AFS.

3.1 Network File System (NFS)

3.1.1 History

The *Network File System* (NFS) was initially developed by Sun Microsystems. The first version (NFSv1) of the protocol was used only within the company, the first public version was version 2 (NFSv2) [PJS+94, p. 137]. Development for this version began in 1984, the implementation was released in 1985 as a part of Sun's operating system SunOS 2.0 [Sat89, p. 152]. As SunOS is based on a UNIX kernel the protocol was strongly related to UNIX system calls. The use of *remote procedure calls* (RPC) as the main communication technique kept the protocol open enough to support clients for other operating systems like PC-DOS [Sat90, p. 212].

NFSv2 soon became a de-facto standard for distributed systems although it had some severe drawbacks: It only supports file sizes up to 4 GB and only operates with synchronous write operations which leads to bad overall performance [PJS+94, p.

137]. To overcome these drawbacks a group of researchers developed a new version of the protocol in 1992 which was released as version 3 (NFSv3) in 1995 [CPS95]. Besides other optimisations it features support for files larger than 4 GB as well as asynchronous write operations. NFSv3 again was widely used in practice, and clients for DOS and Windows have been implemented [PJS+94, p. 138].

A newer version (NFSv4) has been developed recently, first drafts of NFSv4 appeared in 2002. The fourth version mainly enhances security and user authentication besides the support for Windows operating systems [Kir06, p. 52]. This version is steadily gaining market share. As the most often used version of NFS is still NFSv3 [Bra99, p. 2] the following sections will focus on this implementation.

3.1.2 General description

NFS mainly consists of three parts: The server and client implementations and the protocol both use to exchange data. The parts are shown in Fig. 2.

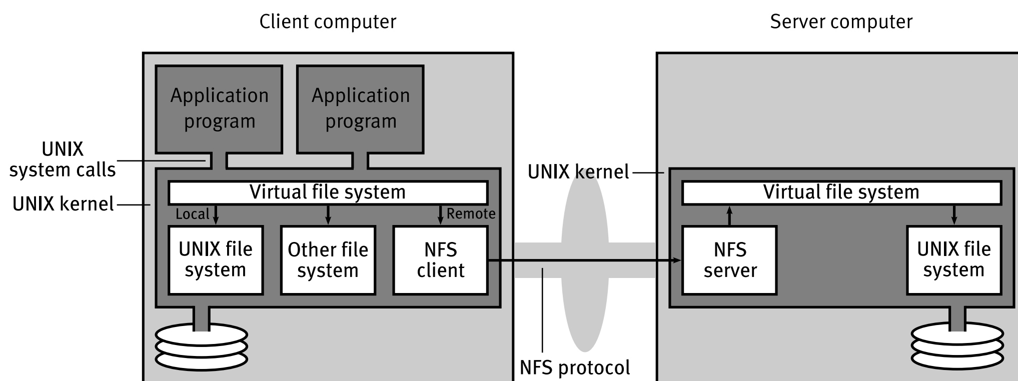

Figure 2: NFS architecture (Source: [CDK01, p. 324])

Overview

The client part is implemented inside the client computer's kernel. Applications can access the module through standard UNIX system calls. For this purpose UNIX features a *virtual file system* (VFS) that allows to mount different file systems within a single hierarchy. If an application performs an operation on a file that resides on a remote NFS server, the VFS module forwards the request to the NFS client module which converts it to an NFS request and sends it to the corresponding server.

Data exchange between client and server is realized by *remote procedure calls* (RPC). RPC provides a "procedure-oriented interface to remote services" ([CPS95, p. 5]), i.e. a collection of procedures that can be called remotely (e.g. over a network) by a client. For NFS, the server module exposes a set of operations that are similar to UNIX filesystem operations. Therefore, a client module only has

to map the incoming UNIX operation to the corresponding NFS procedure before sending it to the server. In NFS, RPC is sent over the stateless User Datagram Protocol (UDP) or the stateful Transmission Control Protocol (TCP), depending on the server configuration.

The NFS server is implemented as a part of the server's kernel, e.g. as a module for modularised kernels.[1] Incoming requests are converted to the related UNIX file system requests and passed to the server's VFS. The result of the operation is sent back to the client [CDK01, p. 323].

File handles and underlying file systems

Besides the determination which file system a requested file resides on, the VFS module is also responsible for the resolution of internal UNIX file identifiers and NFS file identifiers, the so-called *file handles*. File handles in NFSv3 are blocks of up to 64 byte of data that uniquely identify a file on a server. They usually contain a reference to the file system the file resides on, a unique identifier within this file system and a generation number [Kir06, p. 52] [CDK01, pp. 324–325].[2] This file handler allows the VFS to easily determine the file system a file is stored on.

Since the VFS is an abstraction layer that allows access to different UNIX file systems the NFS protocol is not restricted to a special underlying file system [Kir06, p. 52]. Problems may only occur if the used file system does not support unique file identifiers or stores them in directories instead of files [Kir06, p. 53].

Further characteristics

The protocol was designed to be stateless, i.e. "each request contains sufficient information to be completely processed without regard to other requests" ([PJS+94, p. 139]). This allows for an easy failure recovery on the server side: As the NFS server does not maintain states of client requests, no states have to be recovered when the server is restarted. Handling server crashes is within the responsibility of the client module – it simply has to resend a request if the server does not respond [PJS+94, p. 139]. Although the stateless design of the protocol simplifies failure recovery, certain file system features are hard to implement. Requesting a file lock for example can only be achieved by using a separate locking manager [Tv07, p. 501].

A stateless server also poses some restrictions to user authentication. As no

[1] There are also implementations of NFS servers that run completely in user-space; for the sake of simplicity a kernel-space implementation is assumed in this elaboration.

[2] The generation number is necessary for file systems where unique identifiers from removed files may be re-used later.

user sessions are managed, each request to the server has to be accompanied by user authentication information. NFSv3 uses standard UNIX user IDs (uid) and group IDs (gid) to authenticate a user against the server. This mechanism does not provide any security as the validity of the ID is not checked. So an arbitrary user may send requests on behalf of other users by using their uid. This security hole may be worked around by using DES or Kerberos authentication [CDK01, pp. 325–326].

NFSv3 also supports server-side and client-side caching. As the server is stateless all write operations have to be persisted to the underlying disks before a response is sent to the client.[3] Although this design impedes the caching of write operations (*write cache*), an NFS server may cache data that has been persisted for further requests (*read cache*) [PJS+94, p. 139].

NFS clients may cache the results of operations sent to the server to reduce network load. Three different caches can be identified here: A block cache holds block data that was recently submitted, an attribute cache keeps file attributes in memory and a file handle cache allows for a faster browsing within a nested folder structure [PJS+94, p. 139]. To check whether cache entries are valid the client module stores a timestamp with each cache entry. Upon a repeated call to a procedure the module checks whether the client's and server's timestamp match and a refresh interval has not been exceeded. In this case the entry is considered to be valid, otherwise the request is forwarded to the server [CDK01, p. 330].

3.1.3 Classification

NFS widely follows the abstract model proposed in [CDK01]. As seen in Fig. 2 the separation of components is similar to the abstract model, although the distinction on the server side between a directory and a flat file service is not sharp in NFS – an interface for both services is provided by the NFS server module. The counterpart to the flat file system is the UNIX VFS which provides access to the server's storage. The VFS interface is not publicly accessible by clients but can only be reached over the NFS server module.

The operations exposed by the abstract directory service and flat file service can all be represented by corresponding NFS calls. Differences exist in the way files are handled: The abstract model allows to create files without adding them to a directory by using the *Create* operation; the addition is managed by a separate

[3] The only exception is the *asynchronous write* operation which responses immediately and synchronises afterwards when a commit message was sent [PJS+94, p. 139].

operation (*AddName*). In NFS these two operations are combined in the *create* operation which takes the directory where the file will be added as an argument.

In NFS access rights are checked upon every request as the server module is stateless. A hierarchical file system structure is realized within the client module. File grouping is not supported in the sense of the abstract model: File identifiers in NFS do have a reference to the underlying file system (file group), but the group ID is not globally unique. Furthermore the ID cannot be used to determine the location of a file within the network.

General requirements met by NFS

Transparency NFS provides *access transparency* through the UNIX virtual file system model. Applications use the same operations to access files on a remote server as for accessing local files. *Location transparency* is not enforced as clients determine where remote file systems are mounted in the local directory tree [CDK01, p. 333]. This can lead to different file names on different systems for the same file, thus there is no single network-wide name space. Such a name space can only be enforced by configuring all clients similarly [Kir06, p. 58]. As in a single-server setting the server's hard disk is a bottleneck for the overall performance of the network, *performance transparency* is only partly guaranteed. Depending on the network and hard drive speeds clients may not percept any disadvantage when storing a file remotely compared to a local storage. *Scaling transparency* is only ensured as long as a server does not become a bottleneck of the overall system.

Availability As NFS is a stateless protocol the exception handling is completely done by the clients [CDK01, p. 334]. They have to keep copies of data until operations at the server are known to be complete [PJS+94, p. 140]. This is a problem especially with asynchronous write operations. In this case the client has to keep a data copy until the *commit* operation has finished. Handling failures at the server side is as simple as restarting the service. As there are no other measures needed this restarting procedure may as well be automated, e.g. with a monitoring system [CDK01, p. 334]. If NFS is used over TCP and clients handle server failures correctly, NFS provides all requirements for availability.

Concurrent updates Concurrency control can usually be enforced by using file locks. As mentioned before file locks are problematic when using a stateless server. For this purpose NFS uses a separate locking service that is available

via RPC, the so-called *Network Lock Manager*. But as most implementations of this service do not support security mechanisms, arbitrary users may lock files at a server running this service [Kir06, p. 60]. In total, concurrency control can be indirectly achieved in NFS, although the protocol is not the optimal choice in this context.

Replication NFS does not support replication automatically. Duplicate files on different servers will show up multiple times at clients [CDK01, p. 334]. Nevertheless there is a workaround for this purpose using a client-side utility called *automounter*: When a user mounts a remote directory the automounter chooses an arbitrary NFS server from a list of servers that export the wanted directory. Replication between these servers has to be done manually though. This mechanism is obviously only useful for directories that are more often read than they are written, e.g. directories holding program binaries [Sat89, p. 154].

Hardware and software heterogeneity Various client and server implementations for different operating systems and hardware are available [CDK01, p. 334]. Additionally the server is independent of the underlying file systems by using the UNIX VFS layer. Thus NFS can be used in heterogeneous hardware and software environments.

Consistency NFS uses a UNIX-like one-copy semantics, i.e. applications perceive a file as if there is only one copy of it [CDK01, p. 335]. This semantics only works if clients do not cache files locally as this may impose concurrency problems [Tv07, p. 514]. To avoid these issues requests have to be directly forwarded to a single server, which results in bad performance. Thus there is a trade-off between data consistency and performance.

Security As seen in the previous section NFS uses a very weak authentication mechanism by default. Servers only receive the user ID of a requesting user without any proof of identity [Kir06, p. 60]. This allows external users to access an NFS server on behalf of another user which makes accountability impossible. Extensions to RPC exist to overcome this weaknesses, namely Kerberos and DES encryption of user information. Additional security measures are discussed in the literature [CDK01, p. 335], [Kir06, p. 60]. Nevertheless, it can be stated NFS does not support security very well.

Efficiency The literature states that NFS is a highly performant protocol [CDK01, p. 335]. This is mainly achieved by using various caches on the client-side and

fast stateless network protocols like UDP. Anyway, the blocking behaviour of the stateless server may reduce the overall performance of the systems. This is especially true when large files are being written as all blocks have to be persisted to disk before responses to the clients are sent. The usage of asynchronous write operations does improve the overall performance at the cost of additional consistency constraints [Kir06, p. 54].

3.2 Andrew File System (AFS)

3.2.1 History

The Andrew File System (AFS) was developed as a part of the Andrew distributed computing environment. This system was designed in 1982 at the Carnegie Mellon University in Pittsburgh. It targeted at the support of teaching and research at the university [How88, p. 1]. The development of the system started in 1983 [Sat89, p. 157].

The designers wanted to combine the advantages of workstations and timesharing servers. While workstations provide graphical user interfaces and processing power, timesharing servers possess a great amount of storage [How88, p. 1]. These two different approaches were combined in the so-called *Vice-Virtue principle*: Workstations (*Vices*) are connected over a network to timesharing servers (*Virtues*) and are thus able to benefit of each other's advantages. The general principle is show in Fig. 3.

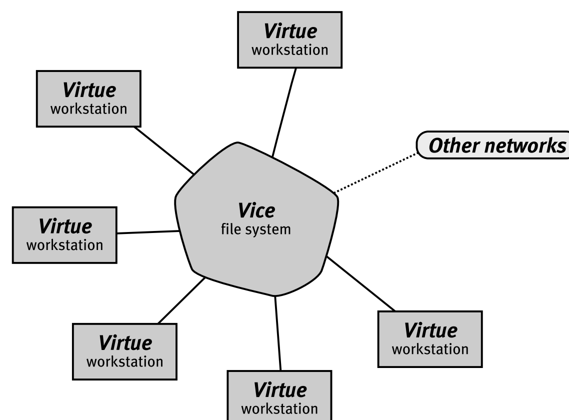

Figure 3: Vice-Virtue principle (Source: [MSC+86, p. 186])

Five general areas the system has to deal with were discovered: The workstation hardware and software, the underlying network, the file system to use, the user interface and a messaging system.The designers chose to build the system on IBM hardware and to use UNIX 4.2BSD as the operating system [How88, p. 1] [MSC+86,

p. 184]. As different Token Ring and Ethernet networks already existed at the campus, the designers created interconnections of the separated networks and built up a hierarchy [How88, p. 1]. They chose TCP/IP as the standard communication protocol to use. The file system (Andrew File System, AFS) allowed users to work at every workstation on the whole campus; details will follow in the next parts.

The introduction of the Andrew distributed computing environment was a success. By the end of 1987, 400 workstations and 16 servers were attached at the CMU, sharing about 6 GB of data [Sat89, p. 157]. AFS underwent several revisions thereafter until development was outsourced to a company called Transarc in 1989 [SS96, p. 203]. Transarc was later acquired by IBM who released the source code of AFS 3.6 under an open source license in 2000. The community continued to develop the file system under the name OpenAFS; release 1.0 consisted of IBM's source release of AFS 3.6 [OA09a]. Client and server implementations have been developed for nearly all major operating systems such as different UNIX and Linux derivates, MacOS X and all recent Windows versions [OA09b].

3.2.2 General description

Overview

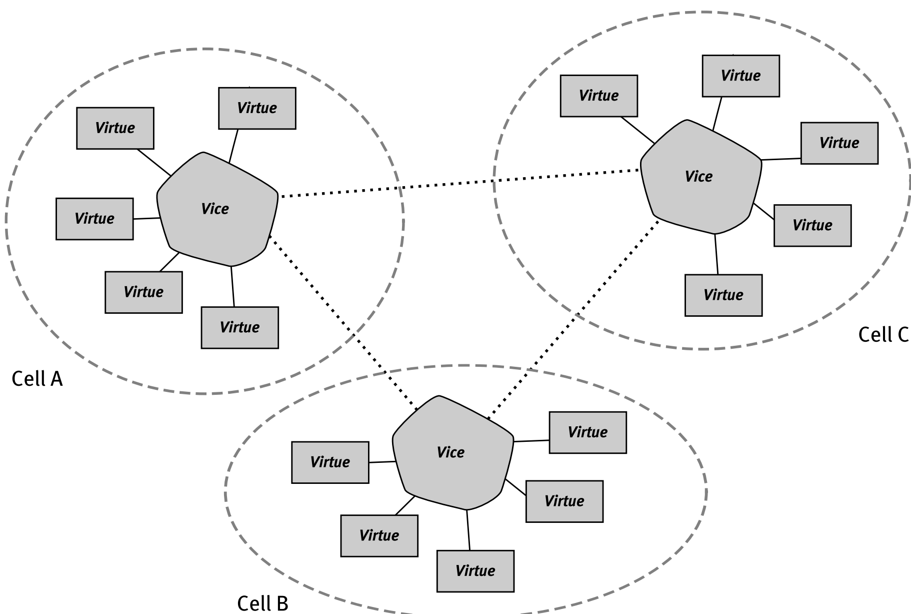

Figure 4: General network layout in AFS

An overview of the general network layout within an AFS network is given in Fig. 4. Computers using AFS are grouped into *cells* – distinct units usually containing some servers and many workstations. These cells can act autonomously and allow for

a better task allocation within the overall network. AFS also supports a cooperation between different cells. In this case clients are able to access files of remote cells in the same way they access local files [SS96, pp. 202–203]. To allow for faster operations on files participating computers make heavy use of a caching mechanism that will be described later in this section.

In AFS a workstation's file system is divided into two parts. The *local name space* contains files that are on the UNIX root partition on a single workstation. It contains files that are needed during boot time or exist only temporarily [Sat89, p. 161]. This small file space is accompanied by a *shared name space* that is common for each workstation. Files in this name space are stored on servers within the network – Vices – that expose a set of file operations and accept requests by RPC over TCP/IP [MSC+86, p. 191]. This name space is often mounted under `/afs/` on UNIX systems. The second directory level usually contains a list of all cells that are accessible within the network, e.g. `/afs/uni-muenster.de/` for a fictional cell within the University of Münster [SS96, p. 204]. In this setting each cell provides one or more *volumes* – sets of files for which this server is responsible. All Vices have a replication of a location database that maps volumes to the responsible server [MSC+86, pp. 190–191]. To satisfy restrictions imposed by most UNIX systems, a local file structure is maintained by using symbolic links from local to shared files, e.g. from `/bin/ls` to `/afs/uni-muenster.de/bin/ls`.

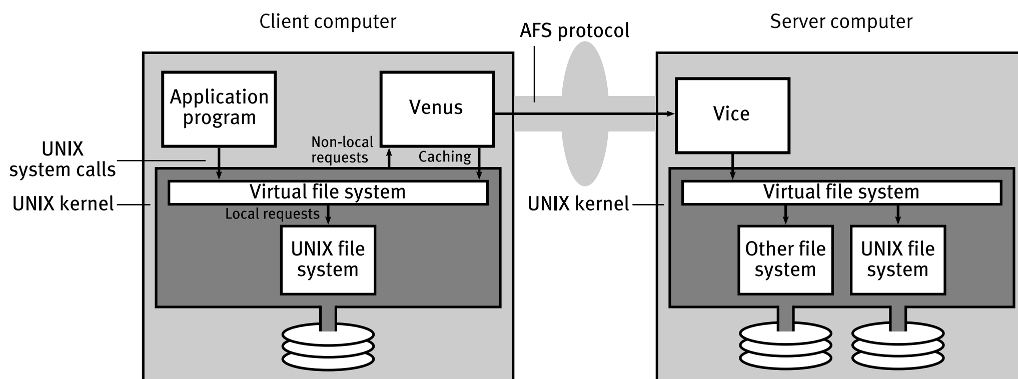

Figure 5: System components in AFS (Source: [CDK01, p.337, p.339]

A part of an AFS network containing one client and one server is shown in Fig. 5. The figure also shows the different software components that realize the functionality of AFS. To handle requests to the shared part of the file system, clients run a software component called *Venus* on their workstation. Venus does not act like a regular file system module within the kernel but runs as a user-space application that intercepts system calls related to files in the shared name space [CDK01, p. 338].

Caching and consistency

If an application requests to open a file that resides on the shared section, the UNIX kernel forwards the request to the Venus service. Venus first searches the responsible server by querying the next Vice's location database. It then checks whether a copy of the file exists in the local cache and whether the copy is still valid – details about the validation method are provided in the next paragraph. If no valid copy exists in the cache it issues a request to the corresponding Vice server. The server returns a file copy to the client's Venus service which places it in its cache. Venus now returns the file name of the cached file back to the UNIX kernel which in turn sends a file handle back to the calling application. After a modification of the file, the kernel closes the local file and forwards the request to Venus. Venus informs Vice about the changes by forwarding a copy of the new file to the server [CDK01, p. 340] [MSC+86, pp. 189–190].

Venus and Vice use a cache validation mechanism that is based on *callback promises*. These promises are issued by Vices to Venuses and can either be *valid* or *cancelled*. While a valid callback promise indicates that the local cache copy of a file is still recent, a cancelled callback promise shows that a fresh copy of a file has to be fetched before continuing. Once a file is requested by a Venus service, Vice issues a valid callback promise back to Venus and logs the access. As soon as changes are propagated to Vice, it notifies all Venuses that have been issued a valid callback promise. The Venuses now change the callback promise's state to cancelled, indicating that the local cache copy of the file is not in a recent state.

File handles and underlying file systems

Files are referred to by 96 bit file IDs that consist of three parts. The *volume number* determines the volume a file is stored in. This volume number is used to query a Vice's location database about the responsible server. The second part contains a *file handle* similar to an inode in a standard UNIX file system. A *uniquifier* is appended to these parts to assure uniqueness of the generated file IDs, similar to the generation number in NFS [CDK01, p. 324].

As the Vice services are implemented as usual UNIX user-space applications they take advantage of the UNIX VFS. This imposes no restrictions on the decision which file system to use to store files. As with NFS the file system only has to support unique identifiers for files similar to inodes and a hierarchical directory structure.

Further characteristics

Server and client implementations in AFS are stateful. Servers maintain states of clients between several requests in form of a list of issued callback promises. This

list has to be maintained even over server restarts. Clients on the other hand have to maintain the states of callback promises as they may be cancelled by servers. Upon a client failure the system has to restore the list of callback promises and check each promise's validity by querying the corresponding server [CDK01, p. 341]. Additionally communication failures, e.g. caused by network downtime, may also lead to inconsistency: Lost promise invalidations lead to situations when cancelled promises are still assumed to be valid. For this purpose callback promises time out after a specific time [CDK01, p. 341].

AFS performs very well for files that are often read and rarely written. In this case local cache copies tend to stay valid for a long time. The same is true for files that are only accessed by a few users [CDK01, p. 336]. The caching mechanism is also well-suited for scenarios with a high locality of reference, i.e. when a file is accessed multiple times in short order (temporal locality). Overall performance in a real world scenario has been measured e.g. in [SS96]. The authors investigated an AFS installation at a university campus and discovered an average file cache hit ratio of about 98% [SS96, p. 214].

The file system does not provide complete one-copy file semantics. It does not broadcast all write operations to all sites holding a copy of the file which would be necessary for one-copy file semantics [CDK01, p.342]. Additionally the system does not enforce concurrency control: As callback promises are only issued during an *open* operation multiple users may concurrently write to the same file. In this case only the last *close* request determines the final state of the file [CDK01, p. 343]. This can be avoided by using file locks.

3.2.3 Classification

The AFS model as depicted in Fig. 5 resembles the abstract model in some ways. The Vice's server module corresponds to the flat file service in the abstract model, while the Venus service realizes the client module. The directory service is partly within the responsibility of Venus, as the translation of path names to file IDs is accomplished by this service.

The interfaces both services offer strongly differ from the abstract model. The flat file service operations *Read* and *GetAttributes* are combined in a *Fetch* operation that returns both file attributes and contents. The *Store* operation combines the functionality of *SetAttributes* and *Write* analogously. Besides some abstract non-atomic operations the interface includes methods handling file locks, callbacks and access control. These method are not part of the abstract model. The directory

service interface is partly realized by the client module. While file ID lookup is handled by the Venuses, the addition and removal of files from a directory is within the responsibility of the server module [Bra99, p. 2] [CDK01, p. 342]. The construction of hierarchical file systems again is a part of the client module, similarly to the creation of hierarchies in standard UNIX file systems.

UFIDs are realized in AFS by the files' identifiers. This identifier partly resembles the abstract model's UFID: The first part containing the volume number can be used to identify servers holding the corresponding file (in conjunction with the shared location database). File handle and uniquifier assure global uniqueness of the generated file ID. File grouping is supported through the shared database that maps location IDs to responsible servers.

Access rights are not only checked during directory lookups but upon every request. This happens due to the fact that directory lookups are only managed within the client module and not within a server. In addition to the standard UNIX access attributes AFS uses extended access lists that are defined on a directory base.

General requirements met by AFS

Transparency AFS provides *access transparency* by intercepting system calls to the UNIX file system layer. Applications do not have to be aware whether a file is accessed locally or remotely. *Location transparency* is also realized, even though cell names occur within the pathes to files by convention. As these names are not necessarily changed when volumes are moved between servers, location transparency is maintained. *Performance transparency* is not fully provided by AFS, as the server is still a bottleneck for overall performance. The heavy use of caching partly overcomes this drawback. As a network of cells can easily be expanded without big modifications, *scaling transparency* is ensured.

Availability As seen in previous sections, AFS supports fault tolerancy only by some additional measures. Inconsistencies can occur when either a server or a client are out of order. Special fault handling mechanisms like a check for cache validity upon restart have to be executed in case of an error.

Concurrent updates Concurrent access is only possible if file locks are used. Otherwise concurrent updates by different clients are overwritten by the last *write* operation. Developers should also consider that locks time out after a specific time and have to be renewed afterwards [CDK01, p. 342].

Replication AFS uses client-side caching, a limited form of replication [CDK01, p. 315]. Server-side replication does not take place as each file belongs to exactly one volume, which in turn is managed by exactly one server. The only exception are read-only volumes containing files needed by many users, e.g. system binaries. These volumes may be replicated to other servers to keep a collection of commonly accessed files on nearby servers.

Hardware and software heterogeneity The file system has been implemented in various operating systems since it was released under an open source license. The implementation in the Linux kernel enables many hardware platforms to act as AFS servers and clients.

Consistency Modifications of a file are not propagated immediately to other parties holding a file copy. Notifications are sent after closing a file and not after a *write* operation was executed. This leads to various situations where data corruption can occur, e.g. when multiple clients access a file at the same time. For this reason AFS does not provide full one-copy file semantics but only an approximation [CDK01, p. 341].

Security The paradigm of low trustworthiness between servers and clients resulted in many well-thought design principles. AFS supports secure authentication via Kerberos as well as encryption. Access control is guaranteed on a per-directory basis. These facts together render AFS a secure protocol.

Efficiency The heavy use of client-side caching is a trade-off between performance and concurrency: AFS provides a high performance at the cost of possible concurrency issues. As [SS96] shows AFS reaches a file cache hit ratio of 98%, so the trade-off seems to be reliable [SS96, p. 214]. As the addition of new cells or workstations to an existing network is easy, AFS scales very well even in large environments.

3.3 Other approaches - the Lustre filesystem

Lustre (*L*inux Cl*ustre*) is a file system that is especially suited for cluster environments. It was designed in 1999, development started in 2002 [Sch03, p. 380]. Thousands of clients and servers can easily be handled by the system while maintaining all requirements to a file system stated in section 2.2 [BS02, p. 50]. This is reached by heavy use of redundancy to avoid bottlenecks [Sch03, p. 380].

Lustre uses a paradigm called object-based storage. Units of data are stored not simply in form of blocks on a hard disk but as objects that offer clients methods for data access, further attributes and security policies. They combine the advantages of low-level block access and a high-level file abstraction. This combination comes at the cost of lower performance compared to block-oriented storage but offers better possibilities for data organisation [MGR03, p. 84].

A cluster using the Lustre file system can be divided into three parts. The first part consists of so-called *Object Storage Targets* (OST). They are handling storage objects and persist them on the underlying storage facility. The *Metadata Servers* (MDS) store metadata information that relate to files on the OSTs. They contain information about directory structures and access rights as well as further file attributes. The third part consists of clients that access the services named above [Sch03, p. 380].

Servers can take one of the three just mentioned roles. They communicate with other servers over a network. Lustre supports various underlying communication protocols, where TCP/IP is probably the most well-known one [Sch03, p. 380]. The core protocol was derived from pre-existing protocols like AFS [BS02, p. 52].

As Lustre follows a different approach than the file systems presented in sections 3.1 and 3.2, the file system only partly follows the abstract file system model introduced in section 2.3. The client module in Lustre is still realized within a user's computer while the directory service and the flat file service are split up. The directory service is realized by the MDSs while flat file operations are handled by the OSTs. Both services provide operations similar to the abstract model, except that file attributes are managed by the metadata service [BS02, p. 53]. Hierarchical file systems are realized within the client module, and file grouping is not considered as files are spread over several OSTs.

4 Summary and Outlook

This elaboration has given a general introduction to distributed file systems. In section 2 general requirements have been derived, leading to an abstract file system model. This model has been used afterwards in section 3, where mainly two examples of distributed file systems have been discussed. The Network File System grounds on a stateless server design and therefore has advantages regarding concurrency control, while the Andrew File System makes heavy use of caching which results in a better performance at the cost of possible data inconsistencies. The Lustre file system follows a far more modularised approach and is especially suited for very

large cluster environments.

The development of distributed file systems is always going on. New computing environments lead to new requirements for distributed file systems. Especially large-scale environments will be in the focus of future developments, for example in the area of cloud computing. With growing amount of stored data further issues regarding data security and privacy may arise that will make a redefinement of some distributed file systems' requirements necessary.

References

[Bra99] Peter J. Braam: *File systems for clusters from a protocol perspective*, Proceedings of the Second Extreme Linux Topics Workshop, Montery, USA, 1999.

[BS02] Peter J. Braam, Philip Schwan: *Lustre: The intergalactic file system*, Proceedings of the 2003 Ottawa Linux Symposium, pp. 50–54, 2002.

[CDK01] George Coulouris, Jean Dollimore, Tim Kindberg: *Distributed Systems, Concepts and Design*, 3rd. ed., Addison-Wesley, 2001.

[CPS95] Brent Callaghan, Brian Pawlowski, Peter Staubach: *NFS version 3 protocol specification*, RFC 1813, Network Working Group, 1995.

[How88] John H. Howard: *An overview of the andrew file system*, Technical Report 062, Information Technology Center, Carnegie Mellon University, 1988.

[Kir06] Olaf Kirch: *Why NFS Sucks*, Proceedings of the Linux Symposium, 2nd. ed., pp. 51–63, 2006.

[MGR03] Mike Mesnier, Gregory R. Ganger, Erik Riedel: *Object-based storage*, IEEE Communications Magazine, 41(8), pp. 84–90, Institute of Electrical and Electronics Engineers, 2003.

[MSC+86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, F. Donelson Smith: *Andrew: A distributed personal computing environment*, Commununications of the ACM, 29(3), pp. 184–201, Association for Computing Machinery, 1986.

[OA09a] OpenAFS: *OpenAFS Release 1.0*, OpenAFS website.
URL: `http://www.openafs.org/release/openafs-1.0.html`,
last access date: 2009-12-20.

[OA09b] OpenAFS: *OpenAFS Releases*, OpenAFS website.
URL: `http://www.openafs.org/release/`,
last access date: 2009-12-20.

[PJS+94] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, David Hitz: *NFS Version 3: Design and Implementation*, Proceedings of the Summer 1994 USENIX Technical Conference, pp. 137–151, 1994.

[Sat89] Mahadev Satyanarayanan: *Distributed file systems*, Distributed systems, S. Mullender (ed.), pp. 149–188, ACM Press, 1989.

[Sat90] Mahadev Satyanarayanan: *A survey of distributed file systems*, Annual Review of Computer Science, 4(1), pp. 73–104, 1990.

[Sch03] Philip Schwan: *Lustre: Building a file system for 1000-node clusters*, Proceedings of the 2003 Ottawa Linux Symposium, pp. 380–386, 2003.

[SS96] Mirjana Spasojevic, Mahadev Satyanarayanan: *An empirical study of a wide-area distributed file system*, ACM Transactions on Computer Systems, 14(2), pp. 200–222, 1996.

[Tan03] Andrew S. Tanenbaum: *Moderne Betriebssysteme*, 2nd. ed., Prentice Hall, 2003.

[Tv07] Andrew S. Tanenbaum, Marten van Steen: *Distributed Systems: Principles and Paradigmsva*, 2nd. ed., Prentice Hall, 2007.