Design and Prototyping of a Secure Internet File System on the Basis of OpenSSH

Norbert Sendetzky

March 7, 2002

Preface

This project was born after I've used several other network filesystems and realized that all these filesystems provide the basic things for file sharing, but real security couldn't be achieved with them. Especially for sharing private data over the internet transparently, there is no filesystem protocol available which someone would dare to use in such a hostile network environment. After thinking some time about how an acceptable alternative must look like and the time to choose a topic for my diplomathesis came closer, I finally decided that it would be great to combine both, my diplomathesis and a project where I can have a lot of fun.

I've been a fan of the Linux system and related open source software for at least four years. I've implemented several other projects on this beautiful piece of software, but until I began starting this work, a project related to the kernel was not among them. So learing something about the implementation of the kernel and digging through the implementation of an operating system was a big motivation for me. Thanks to all people who contributed to the Linux kernel and the Latex word processor. Software is really best when it's free.

Finally I would like to thank Mr. Kern, my professor who supported me a lot while writing this document. And also a big "thank you" to Heidi van der Hor and Nina Scheffer, who helped me fixing grammatical bugs in the final version of this document ;-)

Contents

Chapter 1

Motivation

After reading the title of this diplomathesis many of you surely asked themselves: "Why implementing a new file system? There are already numerous implementations of different file systems and some of them are even useful to share files over a network!"

Sure, but I will show you that none of these fulfills the requirements of a modern file system with regard to security and in more detail authentication and encryption. Especially if you want to exchange data over open networks like the internet and won't miss the comfort you get by integrating external storage space transparently into the file system of your own computer. Then you will realize rather quickly that the security measures in form of authentication provided by file system implementations today are relatively easy to bypass. I also don't know any filesystem which implements strong encryption to protect your data from eavesdropship.

I will show you, why security didn't play a key role in the specification of file sharing protocols we use today and point you to the consequences we are now suffering from. Furthermore I will try to sensitize you for the increasing surveillance of your data traffic either by companies or the government in your country or in any foreign countries.

1.1 History of networking

Remember the time in the 1980s, when personal computer became cheap enough to be deployed widely...

It was the time of a major change in the IT sector when computer devices, which were small enough to be placed on the desktops, began to supersede the huge and complex mainframe machines. Also, the architecture of this devices was far away from being superior, the openness and their good price/performance ratio gave them a major boost in the number of sold units. At first, the personal computers were only used as a better replacement for typewriters and for the spreadsheet analysis. But already at this point and especially in companies, the need arose to share the generated files in a faster way then moving external media from one machine to another to enhance productivity.

This was the starting shot to widespread use of networking technology inside larger companies. Several companies in the IT sector, who also created operating systems, saw their chance to get into a developing market and began to create protocols - partly on top of own proprietary transport protocols - to satisfy the needs of their customers. The protocols which I am refering to are Novell's NCPFS (Netware Core Protocol File System), Sun's NFS (Network File System) and later Microsoft's file system based on the SMB (Service Message Block) protocol. All this protocols were designed to be used in a closed environment such as a corporate network or for connecting two machines at home. Therefore security was only a very small aspect besides low latency and high throughput.

Nowadays, many things has changed. The internet as the standard medium for exchanging data has defined TCP/IP as standard transport protocol and you can reach every computer which is connected to it. At least in theory if the PCs are not protected by a firewall.

The internet is a collection of networks, which are connected among each other. It was designed to be failsafe and therefore every single point of failure must be eliminated. The result of this demand is that there is no single instance which controlls the streams of data passing from one computer to another. Instead the data is broken into small pieces and these packets are routed from one network to another connected network until they reach their destination. The way through these networks is not constant and can change due to a high load on single router or on network failures. Thus you can't guarantee that your packets will not leave their normal path and then be read by someone else. The internet is therefore called an "open network" and considered to be untrusted by default.

In such an environment nobody would dare to set up a file server and share the stored data over a protocol like NCPFS, NFS or CIFS (Common Internet File System)[1], the new name of Microsoft's SMB protocol. Also it has the word "internet" in its name, after a few "blue screens of death" and cracked servers, even Microsoft seems to realize that this protocol is rather useless for usage in the hazardous environment provided by the internet. The reason for this problems are the weak security measures which are implemented into the existing protocols. None of them provides methods for strong authentication which withstands sophisticated attacks. And due to the export regulations on cryptography in the United States until a few years ago, nobody wanted to implement encrypted protocols to protect the transmitted data from eavesdropship.

1.2 Threats in network environments

Let me describe some common threats you will face while using the internet. They can be classified into two major categories:

Technical threats One of the most often used techniques to get transmitted data between two computers is sniffing. It is a passive attack and is therefore on the one hand very easy to do and on the other hand not

detectable neither by the sender nor by the receiver of the message. The only limitation this attack suffers is that the packets have to pass the network interface of the attacker in order to grab them and record their content. Otherwise it may be possible to hack one of the routers between the client and the server to get the packets, but this is much more difficult and implies exploitable security holes in the router software or misconfiguration of it.

Figure 1.1: Sniffing

Another common method to steal data is to attack the computer itself. Due to the complexity of software today, it is common practice that people discover new security holes every day. This can happen in the underlying operating system or in a daemon[1] providing services to users. These flaws could often be exploited by so called "buffer overflows"[2], granting some or all rights the operating system provides to the cracker. Also mistacks made while designing a communication protocol could be used by an attacker to get into a system easily.

Furthermore there is a well known procedure called "man in the middle attack"[3]. The goal is to intercept requests from the client and make an own request to the server with the credentials (username, password, etc.) provided by the client. Afterwards the result is passed back to the client by the attacker. Then, it is easy for the attacker to store all packets which has been passed between these two computers. If these credentials were sent unencrypted or are equivalent to clear text ones, the attacker can use them

[1]The Unix name for a background process which serves requests automatically

for subsequent requests to the server again. This time he may be able to read information, which wasn't transmitted by the first request.

Figure 1.2: Man in the middle attack

At last I just want to mention another attack, whose usage has been increased since the last two years. The goal of this attack isn't to be able to get sensible information, instead it aims at making a service unavailable and is therefore called "denial of service (DoS)" or as a special occurrence "distributed denial of service (DDoS)"[4] attack. They are performed by exploiting weaknesses in programs or protocols and can turn down a computer.

The list of attacks above doesn't lodge a claim for completeness. It only lists a collection of most common attacks performed by malicious users in the internet and sometimes also in corporate or public LANs.

Social threats For the other sort of threats maybe we are together responsible because either they are tolerated by us or we (respectively our politicians) made them by ourselves.

The so called "Law Enforcement Agencies" which is a common term for our police, constantly demand more rights to listen on our private conversations over the internet, radio circuits or other media. Up to now, there is no real analysis on how successful this eavesdropping has been as a whole, at least in Germany and other European countries, and if it was justified in all cases. Furthermore, they are trying to simplify their work by standardizing interception interfaces in communication devices, as described in the ETSI dossiers[6]. This let them get information quicker and without participation of the service providers with less control by others. And don't forget that any

additional observation interface in the communication hardware may open security holes which haven't been seen yet.

Another threat to our privacy are intelligence agencies, which also intercept and record communication or at least parts of it. In former times, during the cold war, these agencies were responsible for supplying information about the enemy, but since 1990, they lost their primary foe. Since then they have been searching for new operational areas and one of these new areas is definitely business espionage[5]. All companies are of course interested in the communications of their competitors because each piece of additional information gives them an advantage. Especially the United States seems to be leading in supplying business information about companies in foreign countries to their own companies.

1.3 Lack of security features

I hope I convinced you that there is a necessity for security in the internet. But how can this be achieved? The answer to this question is more complex: Security is in fact a developing process where both, users and software have to be involved. Besides the parts which requires user interaction, such as training the users and sensitize them about the things they should do or not do to achieve more security, there are two key requirements regarding the software we will have to focus on: Encryption and authentication.

Unfortunately this features aren't implemented into current network file system protocols or the design of them is insufficient. As I have mentioned above, the export of strong encryption was prohibited by the US government until a few years ago. This leads to non-usage of it because the companies don't want to lose a piece of their market share abroad. So neither NFS, NCPFS nor CIFS does anything to protect the content of their data streams. Moreover encryption is a rather expensive operation, which costs a lot of CPU time. Thus either you need a machine with more computing power or specialized chips to achieve the same throughput. Nevertheless

strong encryption is essential to reach a level of security which is acceptable for transmitting data over the internet and we have to cope with it.

Authentication instead doesn't protect your data directly, but it is as necessary as encryption. Its purpose is to prove that you are the one you claim to be. It isn't very nice to sent someone your encrypted personal data and it turns out that the receiver was another person, who stole it. Authentication can be done by exchanging passwords only known by the server and the user or in a more complex and secure way by using a public key procedure like RSA. To make an example, why the authentication have to be improved to current network file system protocols, we may look at NFS and CIFS. NFS provides in its latest version RSA public key authentication, but only between the server and client machine. It isn't checked if the user is trustworthy, instead the server can only be sure that the client machine is known and has to rely on the user authentication of the client computer. This especially leads to big problems on operating systems like Windows 9x or MacOS, which doesn't provide real user authentication. The other bad example regarding to authentication is Microsoft's CIFS, which provides access to server resources by password authentication. But due to the fact that the password digest produced by the hash function is password equivalent[7], this doesn't provide any security if somebody used a sniffer to get this digest.

These were just a few things which are subject to make them better. I hope you understand now, why I strongly disagree if somebody says that the design of another network file system would be a waste of time.

Chapter 2

Analysis

After writing a little bit about my motivation, it should be clear to everyone why I am keen on implementing something new. The basic requirements I mentioned several times are authentication and encryption, but maybe this is not enough.

In this chapter I will analyse existing file system protocols and their implementations for weaknesses, which has been surfaced up to now. I will concentrate my focus on the most commonly used protcols like NFS, CIFS and Novells Core Protocol (NCP File System), but also look at a protocol currently developed by the Carnegie Mellon University named Coda. Sure there are other protocols aswell, but either they haven't been deployed widely and security problems aren't known in detail, or they are similar to the analyzed protocols.

2.1 Network File System

The Network File System (NFS)[8] protocol, originally designed by Sun Microsystems in the late 1980's, is a general purpose filesystem protocol for Unix environments, but the implementation has been ported to many other operating systems aswell. The intention was to create a filesystem for LANs

controlled by an administrator who has administrative control over all machines. Therefore, NFS servers trust the clients to work correct. This has some implications, which will be described later.

First of all a description of the dependencies: In almost all cases the UDP protocol is used as the underlying transport protocol, but in some implementations it is possible to use TCP instead. On top of the raw transport protocol is a layer called Remote Procedure Call (RPC) which implements the different calls for sending, retrieving and modifying data located on the server. It is supported by the eXternal Data Representation (XDR), an additional layer, which evens out the differences between the hardware platforms related to the byte order and implements some basic data types used by the RPC layer. If the UDP protocol is used, the RPC code is also responsible for reimplementing some TCP functionality, like the retransmission of lost packets. All transmissions are sent unencrypted over the wire. The mount protocol is separated from the NFS protocol and not handled by the NFS server. Instead a mount daemon is responsible for accepting mount/unmount requests from the clients and after verifying the validity of these requests it returns a NFS file handle to the client, which grants further access. The verification of a request is rather simple: the mount daemon checks whether the requested filesystem is exported by the NFS server and whether the filesystem is allowed to be exported to the IP address of the client. The NFS protocol respectively the mount protocol describes the possibility to use public key mechanism (DES) or Kerberos authentication, but this isn't implemented into most NFS servers and is not used in the vast majority of LANs, where NFS is deployed. By design, both the mount daemon and the NFS server are dependent on a port mapper, which forwards packets to the ports, where the mount daemon or the NFS server are listening and registered by the port mapper.

There are some aspects of this protocol, which can be seen as design flaws related to the security aspect: In a LAN environment the usage of the UDP protocol is acceptable, but for exporting a filesystem over the internet it is

not. UDP is a connectionless protocol, where packets can't be associated to a transfer between the client and the server besides the ip address. Therefore, it is easy for an attacker to insert packets into the communication of the client and the server, and an intermediate firewall is not able to filter out those malicious packets. For this reason UDP transmissions are blocked by most firewalls unless they belong to the Domain Name Service (DNS) protocol.
Since the packets are sent unencrypted between client and server, neither confidentially nor integrity can be guaranteed, and the authentication is very weak if DES or Kerberos is not used. In addition, this authentication is always done only between the client machine and the server, not between the user and the server. If the NFS server trusts the client machine, then the users on this client computer have access to the whole exported filesystem. Their rights are only limited by the client operating system and if the attacker has administrative privileges on the client machine, he is at least able to read all files and their content not owned by the root account.

Figure 2.1: Sniffing NFS file handles

Furthermore the NFS server can't distinguish between falsified file handles and file handles established by the mount daemon. An attacker can get access to all files which are not owned by root, if he manages to snoop the network and steal a file handle[9]. Some portmappers can also be convinced to forward mount requests directly to the NFS server instead of the mount daemon. Thus the restrictions placed by the mount daemon can be bypassed by a malicious user. The worst case is the chance for an attacker to take over

the server by the installation of a .rhosts file if the filesystem is exported without restrictions and a rhost daemon is running[10]. These scenarios are not unlikely because a NFS request is a simple network message and can be generated by a self written client program aswell.

2.2 Common Internet File System

The Common Internet File System (CIFS), formerly known as Service Message Buffer (SMB) protocol, is an extended version of Microsofts[11] Lan Manager protocol. It was used for file transfers between two computers in DOS in early Windows times. Microsoft extended the protocol by each new Windows version, but kept the original protocol for compatibility reasons. It would be a protocol only used by Windows, if not a free implementation called Samba would exist for all Unix variants.

SMB originally used Microsofts own Netbeui protocol for the message transport, but after it was clear that TCP/IP was winning the competition of the most widely used transport protocol, Microsoft decided (or was forced by the market) to switch to the UDP protocol of the TCP/IP protocol suite. Between the file serving code and the UDP protocol there is an intermediate layer inserted, called NetBios, which is responsible for the translation of the data to the form sent over the network. CIFS uses the UDP ports 137, 138 and 139 for its purposes, like file transfer, remote operations and service announcements. There are eight possibilities how a client can authenticate itself to the server, from cleartext to challenge response mechanisms with hashed passwords. Most of them are offered by the servers for compatibility reasons. The messages sent by the protocol are not encrypted, neither the authentication nor the transport messages.

The same aspects regarding UDP in the NFS protocol section applies to the CIFS protocol as well: it is not suitable for the usage over the internet. Also the complete lack of encryption is a major drawback. It therefore, provides no confidentiality and no integrity, which means that everybody with

access to the packets on their way from the client to the server and vice versa, can read and modify the content of the packet.

Furthermore the server has to trust the client too much and gives away too much information. A server has to accept Null sessions, where no user name and password is provided. Thus a client can at least retrieve a user list and a list of all shares and printers, which are exported by the server.

Figure 2.2: Hijacking a CIFS session

In addition, there are three well known attacks against a CIFS server: SMB Hijacking, Downgrade and Encrypted Handshake Interception[12]. By using SMB Hijacking, an attacker redirects the traffic of a client machine, whose user wants to authenticate himself against a server, to the machine of the attacker. After the user on the client machine has authenticated himself, the attacker sends a disconnect message to the client. Now he is able to access all files on this share without knowing any password.

Another big reason for the weaknesses of the protocol are the old authentication mechanisms offered by all servers for compatibility reasons, which can be used by the SMB Downgrade attack. If the client suggests a low level of security, the proposal of the client is used even if the server suggested a higher level. This makes it possible to downgrade the authentication mechanism to cleartext passwords.
Finally, a design flaw in the protocol helps to decrypt the hashed passwords even if the strongest authentication is used (SMB Encrypted Handshake Interception attack). Passwords are converted to upper case, divided into two halves with seven bytes each (more characters are not used) and padded with zeros if necessary. These strings are then encrypted with DES by using a key with 40 bits. This creates recognizable patterns which eases the decryption by using a brute force attack.

2.3 Netware Core Protocol File System

The Netware Core Protocol File System (NCPFS) is the file and printer sharing protocol used by Novell's[13] network operating system called Netware. It is at least used since version 3.0 and is a core part of the Netware operating system.

In the early 1990s NCPFS used Novell's own Internet Packet eXchange (IPX) protocol as underlying transport protocol, but since version 4.11 TCP/IP is also supported. Novell switched to TCP/IP completely in their latest version (version 6.0) and dropped support for IPX. Now UDP is the protocol of choice for transmitting messages from the clients to the server and vice versa. The Remote Procedure Calls (RPC), which encode the network message, are directly bound to the internal functions of the operating system kernel and not only used for file and printer sharing. A lot of RPC interfaces exist for bindings to the Netware Directory Service (NDS) and other functionality besides file access which are provided by the kernel. The number of RPC interfaces provided by the Netware kernel may have reached several hundred up to now.

There is little known about security holes and vulnerabilities of the protocol, not because the design and implementation is superior, but simply because Novell doesn't talk about security related things in the public. If security holes are found they are discussed in forums of Novell related companies, which develop software for Netware or maintain large amounts of Netware servers of their customers. Available updates for Novell products don't contain a changelog, which documents the fixed security holes, therefore it is best to keep up to date even if the Netware server doesn't seem to be affected by a bug fixed by a new patch.

Because not much information exists about the security issues of NCPFS, which is publically available, I used a document found on a Russian site[14] which describes a lot of security holes and gives advice how to hack the Netware operating system. There are a lot of hacks available, which uses vulnerabilities in the NCP protocol in combination with IPX, but most of them are rendered useless after the switch to UDP. For the usage over the internet the same disadvantages apply to NCPFS as described in the NFS section. All transfers between client and server are not encrypted, so the passed messages can be easily recorded and analyzed. In one of the latest versions of the NCP protocol Novell implemented the possibility to apply a signature to each packet, but it seems that is doesn't work correctly. The transmitted passwords are hashed by a one way hash function, but these hash digests are equivalent to clear text password. Furthermore the hijacking of sessions is possible if the attacker manages to mimic an authenticated client and put client and server out of sync.

The author of the original document[14], which contains the possible hacks mentioned, writes that there seems to be no clear design of the NCP protocol and it has instead evolved over the years. Somebody also called it "a rotten protocol", which is always adapted to the next version of the Netware operating system and up to now never rewritten from scratch.

2.4 Coda

Coda is a rather new file sharing protocol designed by the Carnegie Mellon University[15] and still under development. The major design goals were the possibility to perform operations on the filesystem while being disconnected and automatic reintegration of all changes after the reconnect. It isn't up to now widely used, but seems to be promising.

The transport protocol used by Coda is also UDP like all other analyzed protocols. Above there is a layer which consists of RPC calls and encodes the network messages between the client and the server. These calls only contain messages for filesystem operations and do not include additional functionality like the NCP protocol. The Coda client also maintains big caches of file meta data and file contents for the case of a disconnect. After the connection is reestablished the client automatically replicates all changes to the server if no conflicts arise. The privileges are not managed like by traditional Unix implementations, by using user and group IDs, instead access control lists (ACLs) are implemented.

As the filesystem is rather new, there are no published documents about the weaknesses of the protocol already available. Thus I would like to mention a few basic thoughts: The usage of UDP isn't very useful for the reliable transport of RPC messages over the internet. In a LAN environment, it is acceptable and I mentioned this already in the section about the other protocols. The source code of the RPC layer is rather voluminous, so there may be some security holes found because of the complexity of the RPC implementation. Furthermore there is no possibility to check the integrity of a packet up to now and there is no encryption available by default. The Coda framework provides the possibility to integrate encryption into the protocol, but there seems to be nothing besides a simple XOR operation with a short key available. Maybe there will be a strong encryption algorithm available in the future.

Chapter 3

Requirements

After knowing a little bit about the strengths and weaknesses of the currently used network filesystem protocols, it is about time to define the requirements for a more secure internet filesystem protocol and its software components. This will be done by keeping an eye on the facts we found in the analysis of NFS, CIFS, NCPFS and Coda to avoid their weaknesses and leverage their strengths where possible.

While writing down the requirements, we have to carefully look at the intended usage of the implementation in the future:

- The main usage won't be to get access to file servers in a LAN and transfer huge files to the client. Instead administrators should be able to mount the filesystem of a server or client on his own computer and to do administrating tasks on these machines for example. This may include changing configuration files, copying new packages to the machines and then install them remotely by using a normal SSH login. Furthermore internet service providers (ISPs) should be able to grant customers convenient access to their remote files in a secure fashion.
- It is also not intended as a general purpose filesystem like NFS, which can provide access to whole filesystem trees on startup to all users on a client. The problem of NFS is that it has to trust the clients too

much, which is misplaced in an hostile environment like the internet. Therefore the intended usage of the Secure Internet File System (SIFS) may be best described by making it possible to access files transparently across the internet in a secure way.

- Also it is optimized for a maximum of security not speed and emphasizes a strong client/server model to achive this level of security.

3.1 Transport layer

In this section, I will decribe the requirements for the transport layer of the SFIS protocol built on top of TCP/IP. For further reference about TCP/IP, there is a good book from Kevin Washburn[16] available, which describes the protocol suite in detail.

The transport layer is responsible for a reliable communication between client and server. So because IP is the only protocol of choice for communication over the internet, TCP has to be used. It has some advantages over the UDP protocol, which only forwards all received data packets to the application. TCP is a connection oriented protocol which requires to establish a connection before any data can be sent and a termination of this connection if all data transfers are completed. The reliability of TCP over UDP is achieved by adding error recognition and error correction to the protocol and the retransmission of packets which are lost on the way between client and server. The TCP protocol header includes fields for flow control to manage the data stream as well as sequence numbers to reorder packets if necessary. These sequence numbers are also used to discard packets which are received twice and to request lost packets again. Moreover the TCP/IP stack implemented into the operating systems sends an acknowledgement to the communication partner if a packet was received successfully. Because UDP doesn't contain these features, it is possible for an attacker to inject packets into the communication and only the application can decide whether the packet is correct or not. This caused many network administrators to configure their firewalls to

reject all UDP packets which are not absolutely necessary.

Also the communication between client and server must use a well defined port on the server side. The need to dispatch client connections to different ports if UDP is used is a real nightmare at the security point of view of each network administrator. They have to open a complete range of ports that will be used by the NFS server to communicate with several clients. Also because a firewall can't decide whether the packet comes from a valid and authenticated source or an attacker probing for exploitable security holes without any authentication, the administrator is completely at the mercy of the NFS server implementation.

The ftp protocol has shown the same problem as NFS in the past regarding the use of several ports at once. Ftp uses one port for commands and one for data transmissions. Passive ftp connects to a server and the server replies by announcing a new port where the data connection will be accepted. Active ftp, which is even worse, doesn't repliy with a new port. Instead it uses the port announcement of the client and directly connects to the client. The definition of client and server is therefore reversed in this case and this makes it hard for a firewall to do a good job. Especially active ftp is prohibited by most firewalls completely. To avoid this, only one port should be used by the SIFS server and this port must be used for command and data transfers at the same time. Also the definition of client and server must not be reversed.

All filesystems which use remote procedure calls and especially NFS and NCPFS had shown vulnerabilities over the time. RPC protocols, which have to encode the type, the length and the data of each variable, are more complex than a well defined protocol like IP. Therefore, it is easier to make mistakes when parsing these protocols, which may result in exploitable security holes. NCPFS had a bad history regarding this issue. Furthermore RPC protocols require more memory and CPU time for parsing because of their flexibility, which results in the addressed complexity. Also well defined protocols with fields at fixed positions can be optimized to be aligned to 32 or 64 bit borders

more easily than RPC protocols, which further increases speed on modern processors.

In order to ease implementation and therefore to avoid security holes the protocol should only consist of a simple request/reply scheme. This means that the client sends a request to the server and the server replies only once to the client. The server will never send some data to the client if there was no former request.

The most important thing regarding the protocol is encryption. Without encryption the necessary confidentiality can't be provided. Everybody who has access to the wire or a router, where the data packets are passed through, can read the transmitted messages. The choice of an encryption algorithm depends on two things: There must not be some known attacks which reduce the time to break a used key, and the key must be long enough to withstand a brute force attack. Today algorithms which use a key length of 56 bits are known to be breakable with enough computing power within weeks or months[17]. Thus an algorithm with at least 128 bits key length must be chosen. Recently the American National Institute of Standards and Technology[18] (NIST) finished a competition for a new encryption algorithm. The winner was Rjindael, now Advanced Encryption Standard (AES), which should be used for encrypting the communication between the client and the server. It has proven itself to be resistant to cryptographic attacks and will now be standard for years in many products which use cryptographic algorithms. Furthermore, it uses at least a 128 bit key and there are fast implementations available.

3.2 Server

The server is the most relevant component in a client/server architecture. It is responsible for processing requests from clients and returns a proper response to them. But it is also the component which is exposed to the activities of attackers and therefore a couple of security measures must be

considered.

Figure 3.1: man in the middle attack

To avoid "man in the middle attacks" - one of the possible threats described in the chapter about my motivation - the client needs a validation that the server is the one it claimed to be. Without this validation, an attacker can intercept the clients credentials, connect to the server by using the credentials to authenticate himself and forward the replies of the server back to the client. Encryption doesn't matter in this case, because the client connection ends at the computer of the attacker while the attacker is using his own encrypted connection to the server. The attacker acts in this scenario as a proxy between client and server without knowledge of the client. The usage of certificates signed by a trusted authority is the most commonly used technique to avoid "man in the middle attacks" today. The server sends some informations to the client which contains the address of the server and some informations about the organization running it. These informations are cryptographically signed by the private key of a third party also trusted by the client. The client knows the public key of the third party and checks, whether the signature of this document is valid. If so, the server can be trusted. Another possibility to avoid such an attack is to exchange the certificate, which is now signed by the organization itself by using a secure transfer method. Both methods are acceptable, but at least one should be implemented to secure the authentication.

If the authentication was successful and the user is able to access files, it must be ensured that the user can only access his files. In Unix environ-

ments access to a file is granted by the file permissions associated to every file and directory. Implementations like the NFS server don't care about these file permissions, because the service runs with root privileges and can therefore access the content of the whole hard disk. If an attacker manages to convince the NFS server to give him access to the exported directories, he is able to read all files located there. Thus the better solution is to drop root privileges after the authentication of the user and execute all further commands under the privileges of the user. Then the only files which can be accessed by the user are those which are owned by him or where other users or the administrator granted him additional rights to their files. Furthermore even if the SIFS server module has an exploitable security hole, the attacker can't extend his rights above those of the user.

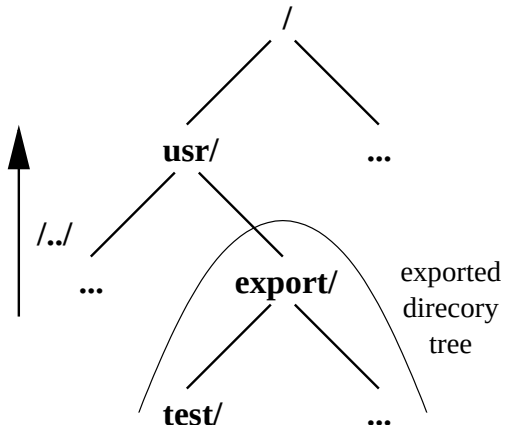

Figure 3.2: Relative path vulnerability

By relying on the file permissions heavily, the administrator of the server is responsible to restrict access as much as possible. Additionally the implementation of the server may contain measures to further restrict access to the files. For example the file permissions of newly created files may be modified to only grant read and write access to the user and not to his group or even to everyone. Furthermore the administrator may be put in a position where he can restrict access only to certain directories on the server in a global mannor like it is possible by NFS or on a per user basis. In both cases the implementation has to care about paths, which contain relative path compo-

nents, e.g. "/../". If this isn't handled correctly, files in the parent directory above the mount point may be accessible and thus the access rules added by the administrator would be rendered useless.

At last the server must not trust inputs sent by a client, neither on the protocol layer nor the extracted strings or numbers. All input should be validated by the server before it is used to avoid exploitable security holes.

3.3 Client

The client is the interface to the user and this makes it to a target of unintended faulty usage or intended attacks. Thus the client has to be a robust piece of software handling all error cases correctly and provide nevertheless the flexibility and security needed.

Like the server, the client must be able to authenticate the user by providing a password, a public key or a certificate to the server. The password exchange is a widely accepted standard mechanism for authentication. The client respectively user, sends a secret string to the communication partner, which is only known by him and the server. This string is then compared to the string in the password database of the other machine and if it matches, the access will be granted. Instead of the clear text string it is also possible to send a hash digest computed from the password and a string received by the server. But this may only be allowed after the communication is already encrypted and the server authenticated itself to the client e.g. by providing a certificate. The public key and certificate method are similar to each other. The difference between them is the kind of validating the identity of the other side. If public keys are used, the server must have already received the public key through a secure channel whereas the certificate must be signed by an authority which is trusted by both parties. The server checks the signature of this third party sent with the certificate by using their public key and grants access if this signature is valid. The authentication with public keys or certificates doesn't require any user interaction and can therefore be used

to authenticate both sides automatically. This is a lot more convenient than the password method and only a little bit less secure. The user must only protect his private key well.

After the authentication is done, the content of the requested filesystem can be exported. Because the user should be able to see the files and directories, he must be able to mount the remote filesystem to a local directory of his choice which is owned by him. This is the task of the client operating system. The contents of this filesystem should be readable or writeable by other users on the client computer if this is desired, but there must also be a possibility to prevent this. A proposed solution for this problem is to map the user and group ID numbers of the files exported by the file system to the uid and gid number of the authenticated user and his primary group. I would like to call this a semantical mapping, because the group of users on the server may not necessarily be the same as on the client machine. Thus it is the job of the kernel to replace the numbers sent by the server with that of the user and his primary group. This enables the user to grant or deny access in a multiuser-capable environment on his specific demands.

Another significant point is the error recovery between the client and the server. In case of a serious error, the communication partners must be able to fall back into a defined state. Such an error condition may be a fatal protocol error. As a result of that, all subsequent packet transmissions may be corrupt from the other side's view, because the begin and the end of a packet aren't known any more. Then both client and server must be able to terminate the connection, but only the client is allowed to reestablish the connection because of the strict client/server paradigm.

The client is responsible for checking the incoming packets for validity in the same mannor as the server. It must not trust the input without checking it. Furthermore the client is also the interface to the user and must thereby carefully examine this input as well to avoid security problems often surfaced in the past. Especially strings must be handled with care. If an attacker is

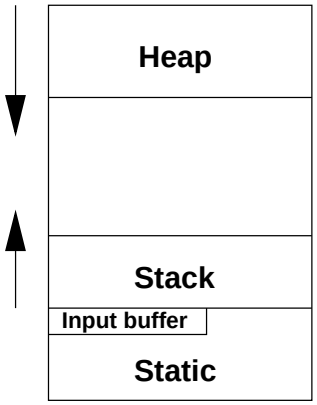

Figure 3.3: Overwrite stack by buffer overflows

able to feed the client with long strings, which overwrite the content behind the input buffer until the return pointer of the current function placed in the stack is reached, he can execute arbitrary code. Therefore strings should be cut in general if the input buffer is out of space.

Chapter 4

Software basis

The SIFS protocol and its implementation as a filesystem driver and a server module are only components, which are useless without a framework they can be built upon. It is necessary to carefully choose the frameworks to ease implementing the components.

4.1 SSH framework

After a little bit of investigation it was clear that an existing framework for authentication and encryption should be used to minimize the effort.

There are two alternatives, which could be used for supplying the underlying services: SSL and SSH. SSL, the Secure Socket Layer is a protocol originally designed by Netcape for the secure retrieval of documents stored on a web server. SSH on the other side was implemented by Tatu Ylonen[19] for a secure replacement of rsh, rcp and telnet, programs which allow remote login and execution of programs on other computers. Both are wide-spread, but SSL more than SSH. The reason, why SSH was chosen over SSL is, because SSH provides a feature rich framework, which eases implementation of new services a lot. Despite that of SSL, SSH and the free OpenSSH implementation already provides a server implementation where new functionality can be added by loading additional modules. Because SSL is only a protocol

description, there is no server available on which it can be built upon. Furthermore SSH provides means to distribute the public keys from the server to the clients on its own, at least in trusted environments. And finally, it also fulfills the other requirements listed in the last chapter.

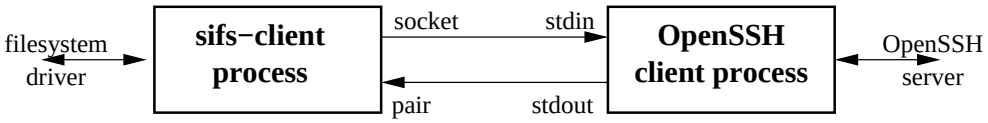

Figure 4.1: Sifsd-SSH client relationship

For transporting the requests from the client computer to server the existing OpenSSH[20] client could be used. Therefore, it is not necessary to reimplement the SSH protocol, saving a lot of time. The OpenSSH client must be executed by program, which has already accepted the connection from the kernel filesystem driver, and must include a set of parameters. Among these parameters have to be the "-s sifs" option to tell the server to execute a server module named sifs after the authentication succeeded. The OpenSSH client is normally waiting for user input on stdin and is writing all received messages to stdout. Thus it is possible to connect a socket pair to stdin and stdout and redirect these socket pairs to the parent process. The parent is then ready for reading requests from the kernel connection and inserting them into the socket pair connected to stdin of the OpenSSH process respectively forwarding replies from the server to the kernel connection.

The server module is a standalone program, which is executed after the OpenSSH server has authenticated the user and forked a child process. Like the user space process on the client computer and the OpenSSH client, these processes are connected by socket pairs for input, output and error messages and the socket end points on the side of the sifs module are mapped to stdin, stdout and stderr. All messages, which arrive over the authenticated and encrypted channel after the connection from the client to the server has been established, are forwarded to the sifs module. The output is sent over the socket connection to the OpenSSH server, which forwards it the other way

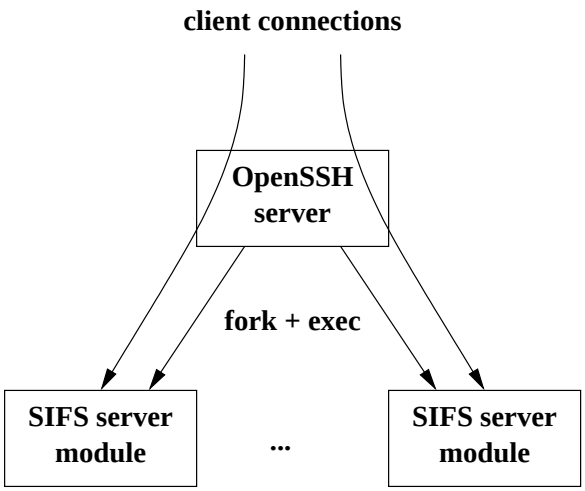

Figure 4.2: SSH server parent-child relationship

round to the client. Only the output of stderr, which consists of textual error messages, is written to a log file on the server and not sent to the client.

4.2 Linux

It became evident that the Linux operating system will be one of the best alternatives for implementing a prototype of the filesystem driver. It provides a Virtual Filesystem Switch (VFS), which is a layer between the filesystem drivers and the file related system calls implemented by the operating system, with a clear design. Currently there are 42 different filesystem drivers included into the official kernel tree and probably more are maintained externally. The number suggests a flexible VFS implementation which matches the need of most filesystems. Also everybody can look at the source code and can learn, how things are done internally to understand the interaction between driver and system call interface. This helps a lot to develop an efficient implementation of a new filesystem prototype. The only disadvantage that comes along with the Linux kernel is the lack of a good and up to date architecture description. This is mainly due to the tendency of kernel developers to document the source code and their aversion to write separate documentation.

Furthermore Linux is relatively wide-spread, at least as server operating system. Especially the target audience (server administrators in companies and internet service providers) often use it as web, file or mail server. They like the stability and reliability of the Linux kernel and the provided security. Therefore it will be the ideal implementation platform for the Secure Internet File System filesystem driver prototype.

A kernel filesystem driver has to be written as a module for the VFS layer. The VFS layer can be seen as a big dispatcher, handling requests from the system call interface down to the appropriate filesystem driver. Its main purpose is to hide the different filesystem types and provide a single interface to all application programs by abstracting from the on-disk data structures for files or directories to a general one, the inodes. The definition "inode" is used as a generic term for all possible file types, like regular file, directories or special files and the VFS layer is responsible for their administration.

Accessing a file (represented in form of an inode by the VFS) stored on a filesystem from an application requires at first the mounting of the filesystem into a directory below the root directory. Only then the content of the filesystem is visible to user space applications. Therefore, the VFS creates a filesystem instance, which is similar to the execution of a program file resulting in a running process instance. With the parameters provided by the mount program through the mount system call, the VFS layer will use the right driver for accessing the filesystem and will show the content under the given directory. It is not important where the filesystem is located, either on a partition on the local disk or on a remote server. This implementation details are hidden by the driver implementation and the VFS layer only act as a mediator, providing a well defined interface to the filesystem driver. A driver can be used several times to access files at different locations with filesystems of the same type and therefore a superblock structure is created by the VFS for each mounted partition (or server directory).

To be able to access the contents of the filesystem, a filesystem driver must provide implementations to the interface stubs of the VFS layer. For example, there are interfaces for accessing files or directories or modifying their contents. Some functionality like the lseek operation (move a file pointer) is the same in all filesystems and by this reason the implementation is done by the VFS layer to reduce the amount of code necessary. Nevertheless this implementation can be overwritten by the filesystem driver if necessary. Interfaces, where no default implementation exists by the VFS layer or where the VFS layer provides only a part of it, must by implemented by the driver.

For a complete (network) filesystem implementation there are a number of such interfaces which must be filled with code by the specific filesystem driver. The tables below contain the names of these interfaces in Linux including a short description, divided into the minimal interface set for a read-only filesystem driver and the interfaces for modifying files or directories.

VFS interface	Description
read_super	Mount the filesystem
put_super	Unmount the filesystem
statfs	Provide information about the filesystem
lookup	Lookup information about an inode
revalidate	Recheck the validity of inode attributes
permission	Check the permissions of an application
readlink	Provide the link content
follow_link	Map the link to the real inode
getattr	Provide the inode attributes
readdir	Provide the names of the inodes in a directory
open	Open a file
release	Close a file
readpage	Read a chunk of the file content

VFS interface	Description
create	Create a new regular file
mknod	Create a special file
mkdir	Create a new directory
rmdir	Remove a directory
link	Create a hard link to a file or directory
unlink	Delete a file
symlink	Create a symbolic link to a file or directory
setattr	Change the inode attributes
rename	Rename a file or directory
prepare_write	Map user data to write buffer
commit_write	Write data to storage medium
fsync	Synchronize buffer with disk
lock	Lock a file or a part of a file

Like mentioned before, the VFS layer implements functionality which is common to all filesystems and it also contains some performance enhancing features. One of them is the dentry cache, a cache which contains the names of files or directories where a lookup of the name and the corresponding attributes were done before. The entries of this dentry cache are necessary for some functions implementing the VFS interfaces to rebuild the path to a file, which should be accessed on the server. Moreover it not only caches existing names, but also the failed lookups as long as enough memory is available. The failed lookups are stored as "negative dentries" with no pointer to an inode structure set.

Everyone can convince himself of the performance enhancement by the dentry cache: the execution of the "ls" command on a directory will be slow the first time. A second and subsequent executions of "ls" on the same directory will return the content within milliseconds, if the revalidation of the directory entries is additionally done in the filesystem driver cache. The effect is rather obvious on network filesystems, where lookup operations are expensive.

For further information about the VFS layer in general, the book[21] from Goodheart and Cox about the internals of the Unix System V is a good reference. For a concrete implementation example, the source code of the Linux Virtual Filesystem Switch is contained in the source distribution of the Linux kernel available at kernel.org[22].

Chapter 5

Design

This chapter contains the in-depth description of the architecture, which consists of three parts, and the protocols relevant to the Secure Internet File System. The design is based on the requirements described in the third chapter while trying to avoid the weaknesses of the analyzed network filesystem protocols in the second chapter.

5.1 System design

5.1.1 Architecture

The starting point for the design was the requirement of a pure client/server architecture, which implies to have different pieces of software on the client and the server machine.

The server must be able to service requests from several clients without interfering with each other. A very secure and commonly used technique by server implementations of all kind in a Unix environment is to spawn a child process for each established connection to a client. The main server process listens to a defined port and accepts new connections from clients executed on machines and connected to the server. Then the server process creates a copy of itself for each connection and listens again to the server port. Meanwhile the child process is able to communicate with the client by

using the previously accepted connection of the server, which was handed over to the child. This child is now responsible for servicing requests for this particular client. If this connection is terminated either by the client or by the child after an error occurred, then the job of the child is done and it finally terminates itself. For a new connection from this client, the server creates a new child not related to the old one.

This approach has some nice advantages over the usage of threads or non-blocking I/O from the security point of view. Each child is a complete process with its own address space. Therefore nothing is shared with other processes, neither with the main server process nor one of the other children. So even if a malicious client is able to force the child to do something which it wasn't intended to do, the attacker can not affect any other process. Furthermore all children can drop all unnecessary privileges inherited from the main server process. This is especially useful if the server listens on a reserved port below 1024, because the process needs root privileges to do this. Without these privileges the potential harm an attacker could cause is drastically reduced. On the other side, forking a child is a very time consuming operation related to creating a new thread or waiting for data sent by one of the clients. Despite all optimizations done by the operating system, it may take five to ten times longer on startup before the first request can be serviced. A thread or non-blocking I/O design may be preferred clearly if speed is a major requirement.

The client is normally a user space program started by the user on the client machine, but not in this case. The goal is to make remote files and directories accessible on the client machine without the need to change any existing program. Because applications use the kernel interface to access all files, the client for the SIFS server has to be a kernel module. This module must be implemented as a filesystem driver translating requests from user space programs into requests for the SIFS server, which are then sent over the network. The reply from the server must also be translated back to a format the user space process is waiting for. In all current Unix implementations a layer called Virtual Filesystem Switch (VFS) is available for a part of this

task. It is responsible for providing a consistent view to all filesystem drivers included into the kernel and often does additional tasks like caching file and directory names.

The first thought may be to implement both, the translation engine and the security mechanisms (authentication and encryption) into the filesystem driverm but this is difficult to do because of the limited functionality provided by operating system kernels. Besides the functionality implemented directly into the kernel and also provided to user space programs, there are only a few additional functions used by other parts of the kernel. In particular functions supporting encryption and authentication are not among them, because they are undesired due to the export regulations of the United States. In addition it isn't possible to access shared libraries in user space, which have implemented such mechanisms. Therefore, the only possibilities are either to reimplement such a library in the kernel or the do all authentication and encryption in a user space program.

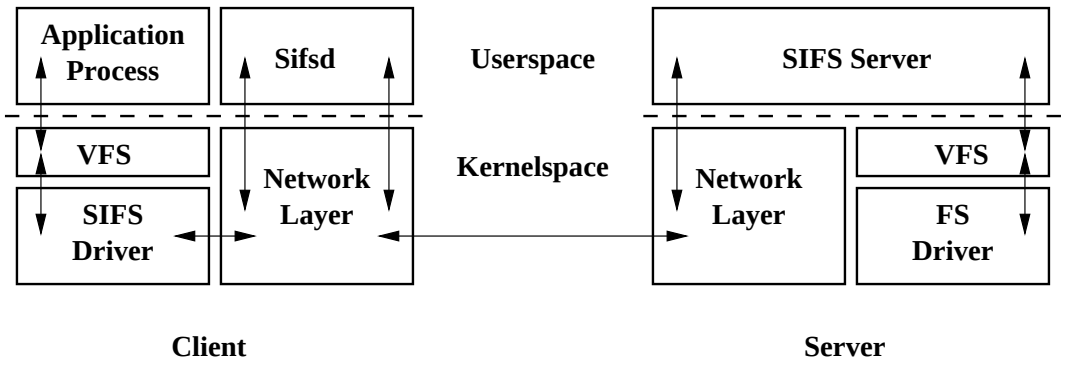

Figure 5.1: SIFS communication model

Due to the complexity of such a reimplementation and the lack of time, the possibility to reuse existing code in user space seemed appealing. Also Jeff King[23] suggested the usage of a daemon running on the client machine and listening to the loopback network device for requests sent by the filesystem driver to the server. This daemon, which will be called "sifsd" or client daemon, is responsible for establishing an authenticated and encrypted con-

nection to the server and sends the packets back and forth between kernel and server. Besides the advantage of reusing code, the filesystem driver is now more likely to be accepted in the official kernel distributions.

The disadvantages are a slight loss of security on the client machine and a performance decrease. Because all transfers between the kernel and the sifsd are not encrypted, the super user is able to attach a packet sniffer to the loopback network device and to read the information passed to one or the other. But due to the privileges of the super user, he may have simpler methods reading this information. All other users are unable to access it because they are not allowed to sniff the loopback network device, and the packets won't leave the computer on this way. It isn't known yet how big the impact on the performance is, because of the sifsd. In the end the data is additionally sent and received twice, once from the kernel to the sifsd and another time from the sifsd back to the kernel, before it is finally sent to the server.

5.1.2 Description

Now, after the basic architecture is clear, a more detailed description of the single steps done by the client, sifsd and server follows. On the basis of activity diagrams the different types of actions of these components as well as the possible error situations will be shown.

The sifsd, acting as proxy between the kernel code and the server, is responsible for forwarding all high level requests back and forth. Before it can perform this task it must establish a connection to the server which is encrypted and authenticated (see figure 5.2). For authentication there are at least two methods which have to be supported: Password and public keys. For password authentication, which is probably used in most cases, a username and password must be sent to the server to prove the identity of the user. These credentials are requested by the client program the user executes and therefore known by the sifsd. In fact it won't be necessary to hand

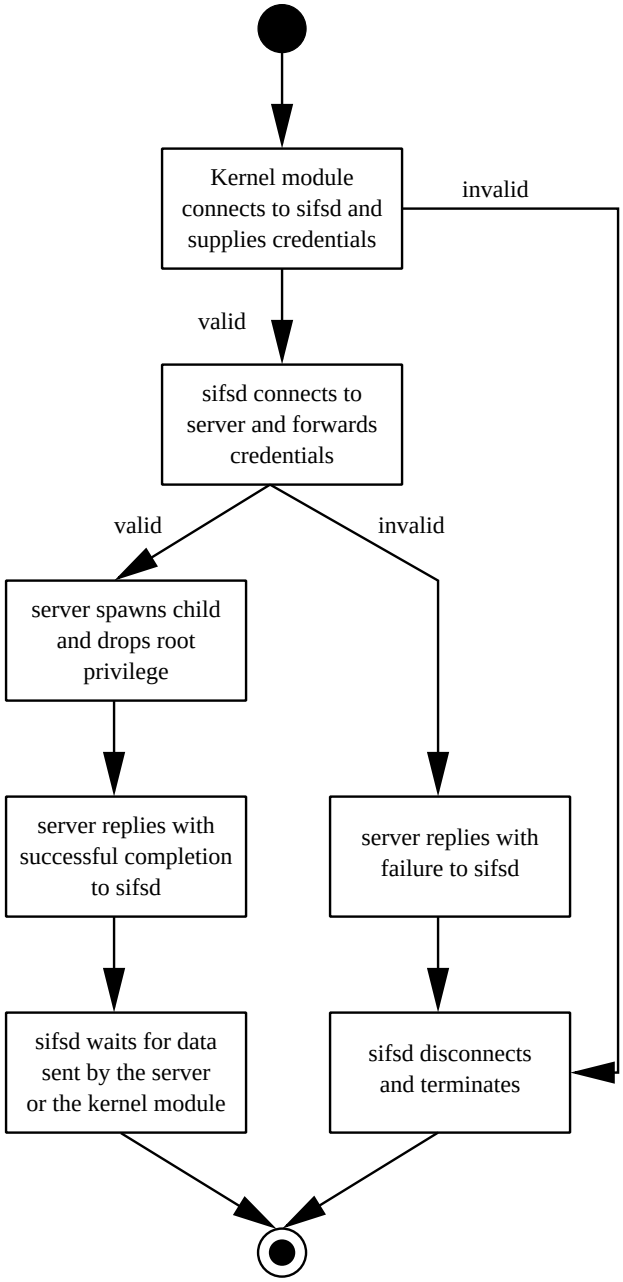

Figure 5.2: Connect activity diagram

them down to the kernel filesystem driver, because the sifsd is already able to establish a connection to the server, but the sifsd itself has to wait for a connection from the filesystem driver to initiate the connect procedure to the server. In order to prevent an attacker from connecting to the sifsd and sending requests to the server, the credentials are transmitted by the connect request from the kernel to the sifsd. These credentials are then compared to the original ones and only if they match, the connect sequence to the server is initiated. The server also has to check the credentials against its database and if they are valid, the server spawns a child of itself for the new connection. At the same time this child drops its privileges inherited by the server process to limit the access of the user. In both cases - the credentials are valid or not - the server sends a reply to the sifsd. This reply is then forwarded to the kernel and if it was a negative one (an error message) the sifsd terminates.

There are two possibilities for the kernel filesystem driver to disconnect (figure 5.3): If a user wants to unmount the filesystem and remove it from the client filesystem tree, and if the kernel detects a fatal error regarding the protocol. A fatal protocol error may be if the connection screwed up and the received replies from the server can neither be dedicated to former requests nor the payload makes any sense any more. In the error case, the kernel terminates the connection and the sifsd does the same to the server, but the sifsd must remain active and has to wait for a new connection from the kernel. In the other case, when the user triggers a normal shutdown, the sifsd has to terminate the connections as well and must then terminate itself. To differentiate both cases, the kernel filesystem driver will send a disconnect message to the sifsd if it should cleanup and exit, respectively terminating the connection to the sifsd. Without this disconnect message the kernel signalizes that a new connection will be opened soon.

A reconnect in case (figure 5.4) of an error is quite simple. It is only a sequence of a disconnect/wait operation, which affects the connection to the sifsd, followed by a new connect to the sifsd. After the sifsd knows that the channel to the kernel filesystem driver was closed, it closes the connection to

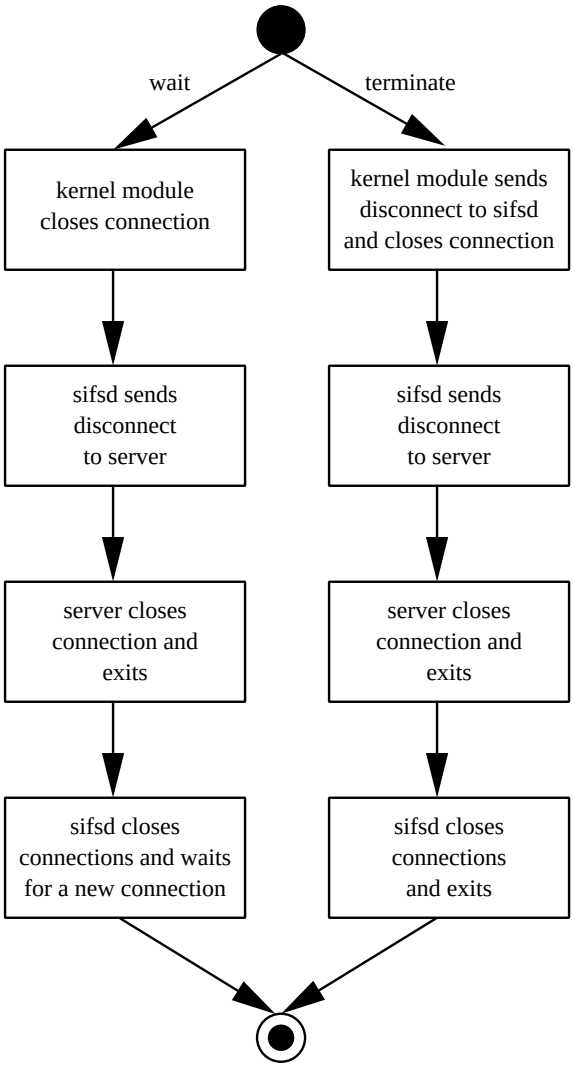

Figure 5.3: Disconnect activity diagram

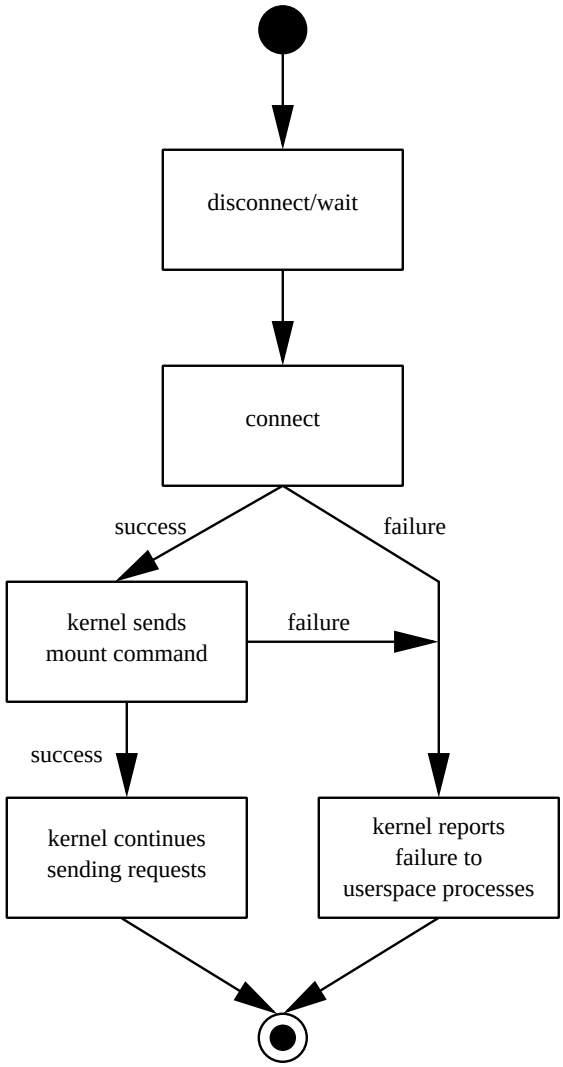

Figure 5.4: Reconnect activity diagram

the SSH server and waits for the new connect by the kernel. The credentials, which are sent by the kernel, must be validated again before a new connection to the server can be established.

After the explanation of the basic operations, only the steps involved in mount, unmount and data transfer operations are left. They can also be best explained by activity diagramms.

The mount request (figure 5.5) to attach a remote filesystem to the filesystem tree is triggered by the user. Therefore he has to execute the sifs-client program, which includes the sifsd server, and the mount command. The sifsd listens to a given port on the local loopback device while the mount command supplies the credentials, the IP address and the port where the sifsd is listening and the local and remote directories, to the kernel. The connect operation followed by the mount request of the SIFS message protocol decides about the success or failure of the mount process. The result is handed to the mount process in userspace, which reports a successful completion or the error message to the user. On success the sifsd remains active and forwards the requests and replies.

An unmount request (figure 5.6), which removes the remote filesystem from the client filesystem tree, can also only be triggered by the user. For this kind of action there is also no special tool is provided because it can be handled by the standard unmount program. Umount does a few checks to the supplied parameters before the kernel is requested to remove the given directory tree. This can only be done if none of the files or directories are already in use by another program. If there are open files or directories the operation fails, the connection remains active and an error message is returned to the umount process. Otherwise an unmount request is send to the server, which signalizes to clean up and terminate the server process. Furthermore the connection to the sifsd will be terminated with a disconnect message and the successful completion reported to the umount program.

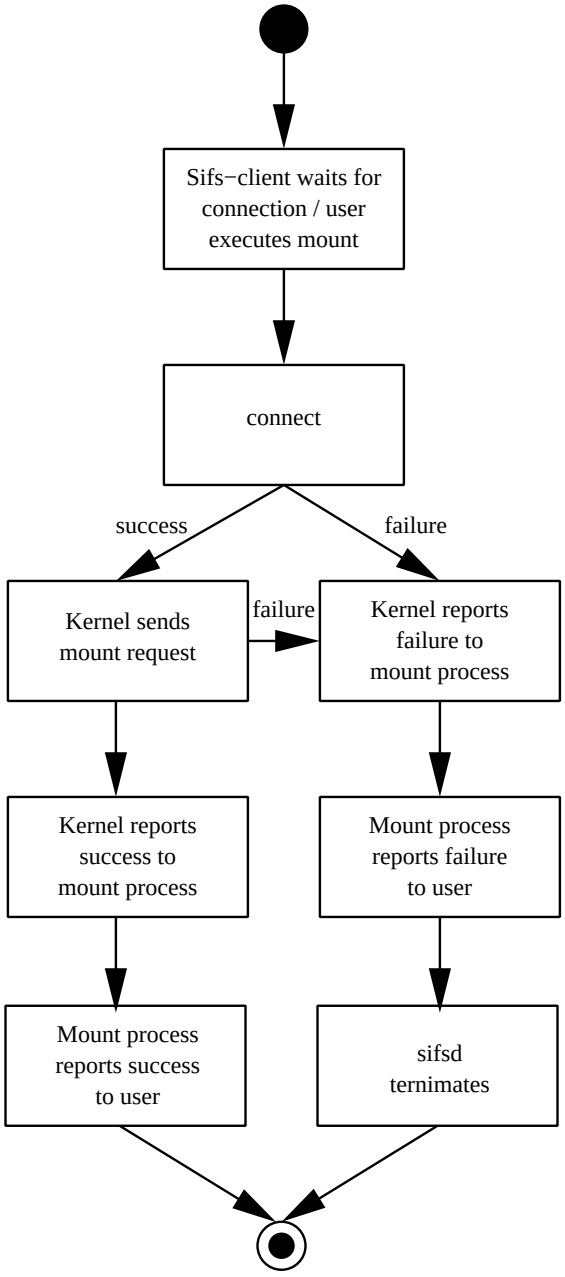

Figure 5.5: Mount activity diagram

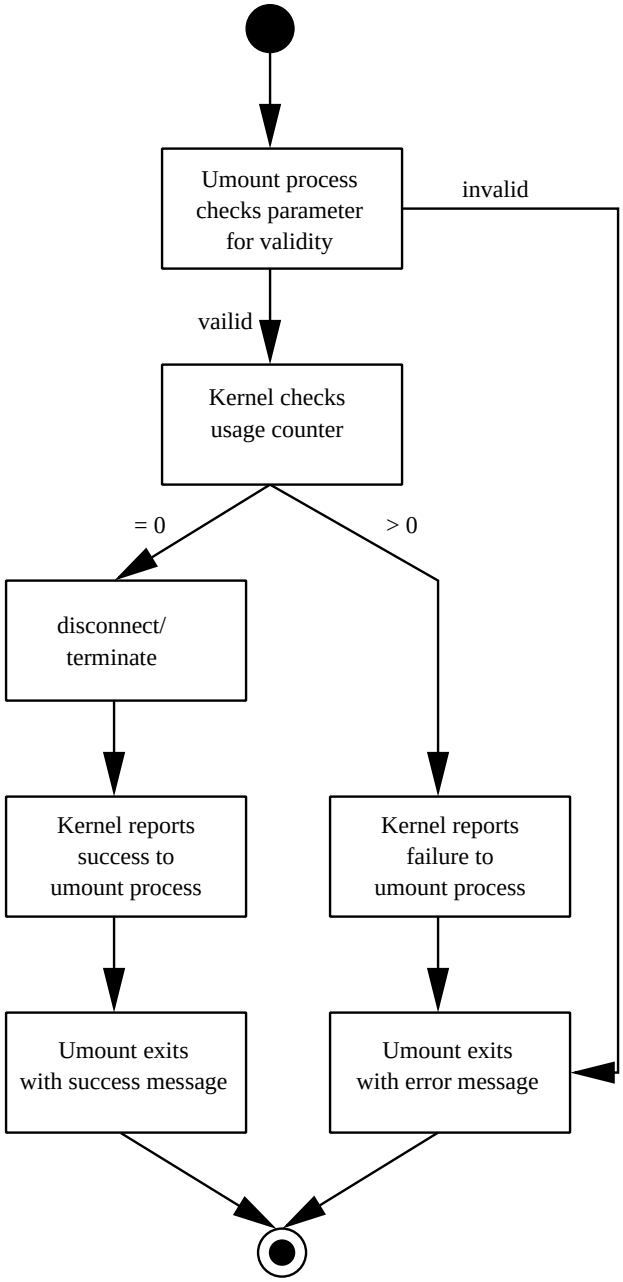

Figure 5.6: Unmount activity diagram

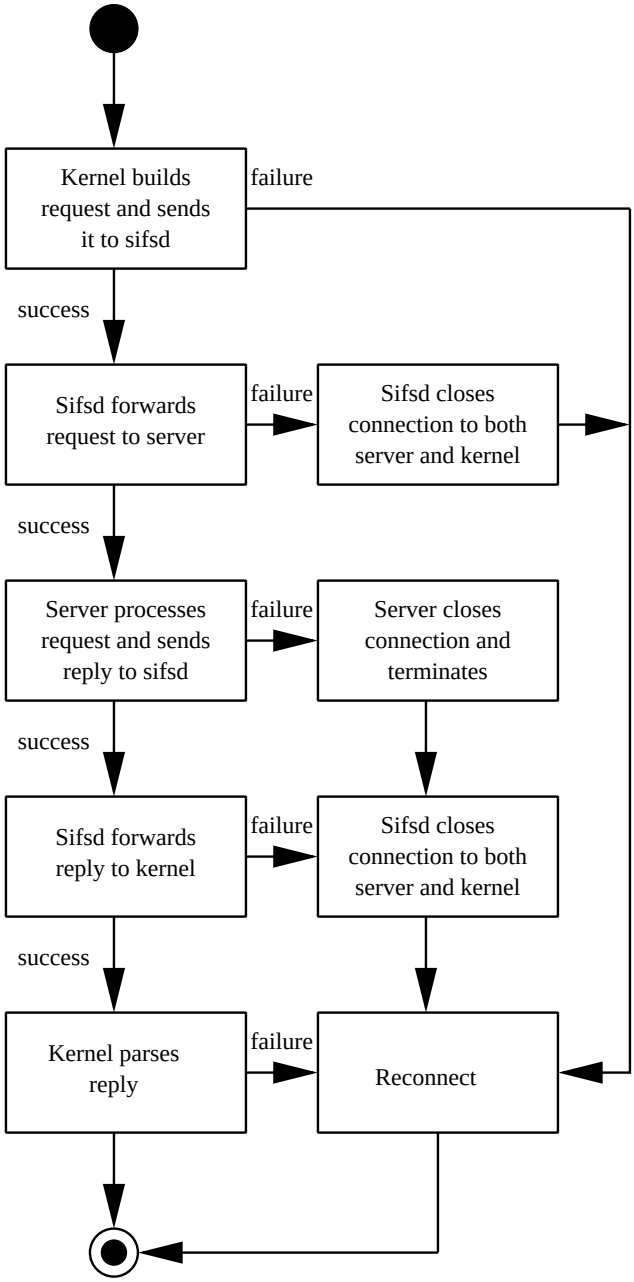

Figure 5.7: Data transfer activity diagram

The last and most often used operation handled by the sifsd is the transfer of requests and replies between the kernel filesystem driver and the server (see figure 5.7). The requests are built by the kernel filesystem driver and are sent to the waiting sifsd. The sifsd forwards it over the established connection to the server, which tries to process the requested operation and builds a reply packet. This reply is then sent back to the sifsd, which is responsible for relaying it to the kernel. The filesystem driver will parse the reply and make the result available to the waiting user space program. On this way there are numerous possibilities for errors, which may occur. If the kernel can't send a request to the the sifsd, it always tries to do a reconnect, and the kernel will report an error to the user space programs that the data currently isn't available. On the other side, if the server closes the connection because of an unexpected error, the sifsd also terminates the connection to the kernel and doesn't try to reconnect itself. The sifsd will stay alive in both cases, the termination of the kernel or the server connection.

5.1.3 Problems

The proposed solution described above poses some problems which are also known by other network filesystems. Mainly there are at least three different kinds of problems, file locking, file permissions and the time difference.

The file locking problem arises in scenarios like this: A client program requests a lock on a file stored on the server. The connection is established and the lock is granted by the server. If a fatal error occurs and the connection must be terminated and reestablished, then the server doesn't remember the locks granted to the client any more. Meanwhile another process on the server or another client may have requested a lock on the same file. It is granted, because the server process servicing requests for the first client has exited itself after the connection was shut down. The application on the first client is sure it holds the lock, but in reality the second client got the lock before the first one reestablished the connection. Network filesystem protocols like NFS doesn't face this problem to this extent because of the

connectionless nature of UDP. Furthermore NFS uses a lock daemon for all locking requests, which is separate from the daemon servicing file requests. Admittedly the lock daemon must have root privileges to handle requests from all users, which makes it to another potential target for remote attacks.

A solution compatible to the requirements would be a separate lock process like the one NFS has, but one spawned by each server child process. The child process is connected to the corresponding lock process by named pipes for example and has the same privileges like the child process. As both processes have only the privileges of the authenticated user, they can only access the files where access rights are granted to them. This limits the potential harm they can do, if one or both of them are exploited after a successful attack. The main purpose of the separate lock process is that it remains alive for some time in case of an unclean shutdown. If the connection to the client is closed before an unmount is done, the server child process terminates while the lock process waits for some time. If a reconnect from the same client occurs, the new child process of the server daemon looks for an already existing lock process. If the lock process still exists and has saved the states of all active locks, the new child reestablishes the connection through the named pipe and can proceed servicing request from the client. The locking state on the client became not invalid in the meantime. The reason why the lock process terminates after some time if no reconnect occurs is that an attacker should not be able to exhaust the resources of the server by terminating connections and not doing a reconnect. This would lead to many lock processes that pollutes the process space on the server. There is no need for a description of the protocol between the server child process and the locking process, because it is only an internal interface not seen by the client. Both processes are part of the server software and each implementor is free to use his own solution.

Another issue is the problem regarding the file permissions on the client and the server. It can not be presumed that the user on both systems has the same user and group ID. Therefore there must be some kind of translation between

client and server to ensure the right mapping of the server user/group ID to the client user/group ID. Otherwise, if only the raw numbers are displayed on the client, there will probably be some clashes with existing users or a huge amount of unassigned IDs. The mapping of user and group IDs can be done on the client to relieve the server without weakening security. It prevents application processes on the client to access files on which they should have no permissions, but it has to be done in the kernel by the filesystem driver. There is no alternative beside trusting the driver. If an attacker manages to insert a modified driver into the kernel, he can change the mapping and the displayed file permissions. Then he might be able to access all files on the server on which the user who mounted the filesystem also has access rights, but for this attack root privileges are required on the client machine and it is impossible to build a secure way for file access through a mounted filesystem if the user can not trust his own computer. If he can trust the client machine, then it is possible to limit the access only to the user by clearing the permissions of the group and the other users in the parent directory. This prevents them from even seeing the file and directory names of the mounted filesystem.

The server should export the files with the original file permissions to allow the user to give other users on his computer access to the files as well. The only problem is the correct mapping of the IDs. The solution is a direct mapping of the user ID on the server to the user ID on the client for all files, where the user is the owner of these files. Futhermore, the group ID should be mapped from the group ID of the user on the server to the group ID of the user on the client. I would like to call this a "semantical mapping", because the users who belongs to the group on the client are probably not the same as the users who are members of the group on the server. This kind of mapping should be done because there should be a way to enable the communication between the users and not to eliminate it in general. On the other side, if a file isn't owned by the user who authenticated himself on the server, its user and group ID should be mapped to root. This mapping is probably next to the original situation regarding the privileges. There is only one situation, where the user can't access a file despite the fact that in reality the server

would grant access. If the authenticated user belongs to the group attached to the file, but the group isn't his primary group, then he is unable to access the file. This is not nice, but I think it is not too bad. In all cases, the file permissions (read/write/execute) displayed on the client must be the same as this one on the server.

The hardest problem is to find a good solution for the time difference between the client and the server, because programs like "make" needs the modification time of a file to work correctly. Even if the hardware clocks in both computers were once synchronized, they tend to differ after some amount of time, if they are not frequently adjusted. This can only be done in a reliable way by an external radio signal. It is very unrealistic to demand that everybody who is using the SIFS must have a correct system time. The time can also be distributed over the Network Time Protocol (NTP), but this isn't a solution, which is 100% secure, because the packets can be forged on their way to the client. From this point of view the solution may be to add or substract the time difference on the client. It turned out that this is also a difficult task, because the time difference must be measured at least once after the connection is established. This would require a separate protocol, which would complicate the design a lot and would raise further problems. One of these problems is to measure the transit time between the client and the server. Another problem would be how attacks should be detected and handled, if an attacker deferes the arrival of the packets responsible for time measurement.

There is no known secure solution which is applicable in this case. Therefore, it may be for the best not to modify the access, modification or creation time, which are sent by the server. The client may change the time values of the files on the server, but then it has to use its own system time. At least this doesn't make the situation worse.

5.2 SIFS message protocol

In order to exchange data between two computer or only two processes we need a description how this data is transported. Otherwise no one would understand, what the other process or computer is talking about. To be successful a description of all commands known by this protocol and a detailed layout of all fields included in the commands is necessary.

The protocol described on the following pages is not specific to the topic of this diplomathesis, the secure internet filesystem. Instead it is designed to be a general purpose transport protocol to exchange data related to filesystem operations efficiently. Also it is not bound to a specific operating system, but I have used Unix and especially Linux as a source of how things could be done and which atomic operations are neccessary. As far as possible my design is compatible to POSIX standards and is abstract enough to be easily implemented in different operating systems.

Furthermore the filesystem protocol is reduced to the basics, so it can be used in a lot of ways. For the secure internet filesystem the parts of authentication and encryption are done by the OpenSSH implementation. As underlying transport protocol the SSH protocol over TCP/IP is used, but like I said before this separation enables any implementor to build other components around it and use other underlying transport protocols as well.

5.2.1 Basic thoughts

As I mentioned above, the filesystem protocol is designed to be POSIX compliant where it is possible and reduced to the basic operations. Only where the POSIX standards doesn't describe functionality needed or it violates the "only atomic operations" principle, I will do things in another way.

Unlike the NFS protocol, which is built upon remote procedure calls, the concept behind this one is the same as used by the IP protocol. Each packet consists of a well defined set of fields which are arranged to utilize the simi-

larities of the different commands which simplifies parsing a lot. Each packet includes a header part, which is the same in each request and reply. Only the values change. All fields after the header are called payload and the composition may differ a little bit for each command. The payload consists of the parameters needed by the server to do its work, retrieve the results and send them back to the client. Of course, the reply can also contain a payload like the read operation for example, but it is not sent to the client if an error occurs.

Another thing I kept an eye on was the fact that all fields are aligned to 32 or 64 bit borders where possible, like in the IPv6 protocol. Modern CPUs can in this case perform the operations at higher speed due to faster load/store operations between memory, cache and processor registers. The disadvantage is an increased resource (memory) usage because of the fact that some fields are larger than needed. Fields other than strings, which are a concatenation of bytes, are set in network byte order also known as big endian format.

A very important part regarding the usage of the protocol in this case is that only the client can send requests. The server itself only handles requests from clients and replies to them, but will never do something without a former request.

5.2.2 Header

The header of each request consists of three fields: the id, the size and an opcode field. The header is only 12 bytes long and thus increases the whole packet moderately.

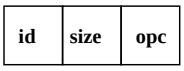

Figure 5.8: request header

The reply header is the same as used by a request, but replaces the opcode by an error code.

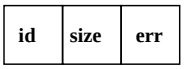

Figure 5.9: reply header

type	description	length in bytes
id	unique identifier	4
size	total size of the payload	4
opc/err	command/error code	4

The first field, the unique identifier, contains an integer which makes it possible to associate a reply to a former request. If a reply has the same id as a request sent before, we know that they belong together. The easy way is to increment the id counter by one for each request. But it is also possible to implement a different scheme, which provides a more secure way to avoid attacks. If all ids were used by former requests, the id counter must be reset and can be incremented again by each request.

The length of a packet is specified by the second integer, which is also 4 byte long. Theoretically, a packet may be 4 GByte long. Thus very big data transfers can be handled at once, but due to the limited size of memory and limitations in the underlying protocols a value around 8 KByte is a good size. This has also been proven a good value by various NFS performace benchmarks, but it is only a recommendation, and implementors may change the buffer size of the client and the server to match their needs. The size includes the length of the whole payload and excludes the header, which is always 12 byte long.

The opcode/error code field is the last one regarding the header of a request or reply. If it is part of a request, it contains the operation code shown in the figures where the different types of requests are described. The value of the opcodes are always greater than zero and must match one of the known opcodes. The operations and therefore the numbers follow a simple scheme:

they are divided into operational areas. Values between 1 and 255 belong to the filesystem operations like mount and unmount whereas values bigger than 256 belong to file or directory operations. This scheme will be used to enable implementors to build fast and simple implementations, which can utilize tables of pointers to branch to the parsing functions.
If the header is part of a reply, then this field contains the error code sent by the server. A value of zero is equivalent to no error respectively a successful completion of the requested operation whereas a value smaller than zero (IEEE representation of a negative value) represents a certain kind of failure. Both, opcode and error code must be sent in network byte order over the wire.

5.2.3 Strings

The strings utilized by this protocol are used to reference the location of files and directories on the server.

Figure 5.10: string representation

type	description	length in bytes
c	number of strings, which will follow	2
s	number of bytes the next string contains	2
string	a zero terminated string	n

There are only a few rules as to how strings are represented. They are always located at the end of a packet because their length vary. Thus all other fields of this packet are at a fixed position, which simplifies parsing. Furthermore the outline of one string is only a special case of many strings. The description of strings always begins with the number of strings, which will follow and is at least one. Then the strings itself are described by the length (including the trailing zero byte) and the content of the string. A string always ends with a zero byte like in C. Thus a string sent to the

communication partner can be used directly as input to system calls. After the first string more strings may follow which are again described by their size and their content. The maximum possible size of one string (content and trailing zero byte) is limited to 1024 bytes. This is a limitation set by the POSIX standards.

5.2.4 Error codes

Every reply to a former request contains an error number, which is stored in the header. An error code of zero is equal to a successful completion of an operation, error codes which are less than zero are equal to a failure. As you may see in the table below, the error codes are similar to that which are used by Unix, but divided into error classes to reduce their number. Therefore not all error codes known by POSIX or other Unix related standards are used. Instead they are mapped to one of the error classes.

Error code	No.	Description
SIFS_EOK	0	Operation successful
SIFS_ESERVER	-1	Internal server error
SIFS_ENOTSUP	-2	Operation is not supported
SIFS_ENAMETOOLONG	-3	Path or filename was too long
SIFS_ENOENT	-4	File or directory is unknown
SIFS_EEXIST	-5	File or directory already exists
SIFS_ENOTEMPTY	-6	Directory is not empty
SIFS_EPERM	-7	Operation denied permanently
SIFS_ETEMP	-8	Operation temporarily not allowed
SIFS_EACCES	-9	Operation denied due to missing rights
SIFS_EINVAL	-10	Argument was invalid
SIFS_ENOSPC	-11	There was no more space available
SIFS_EIO	-12	An I/O error occurred
SIFS_EPROTO	-13	An protocol error occured

SIFS_ESERVER: An error which was not recoverable occured on the server, like an out of memory condition

SIFS_ENOTSUP: The operation is not implemented, because the underlying system doesn't support it

SIFS_ENAMETOOLONG: The underlying filesystem only supports names (strings) with less characters

SIFS_ENOENT: The string contains a name, which doesn't exist in the path to the file, or the file or directory itself doesn't exist

SIFS_EEXIST: The name already exists at the same position in the filesystem

SIFS_ENOTEMPTY: A directory, which should be (re)moved is not empty

SIFS_EPERM: The permission to perform this operation is denied permanently. For example a file which should be modified resides on a read only filesystem

SIFS_ETEMP: Due to another process or operation the requested operation can not be performed yet, but will be possible in the near future

SIFS_EACCESS: You do not have the permission to perform this operation on this file or directory

SIFS_EINVAL: The file is not suitable for this operation or the argument was out of range

SIFS_ENOSPC: There are no more free blocks on the filesystem or no more inodes available

SIFS_EIO: The operation could not be completed, because the hardware was not able to perform this operation correctly

SIFS_EPROTO: A protocol violation was detected while processing the input stream, e.g an invalid opcode or no filesystem is mounted up to now

5.3 Kernel-sifsd transport protocol

By the desicion to source out the SSH stuff including the cryptographic algorithms into userspace, there is a need for an additional transport protocol between the kernel of the client operating system and the client daemon (sifsd). This protocol is responsible for exchanging the parameters used by the client daemon to establish a connection to the SSH server as well as forwarding the requests generated by the kernel and returning the replies from the SSH server. It is also the only way to let the client daemon know, when it should disconnect and terminate itself at the end of the session.

5.3.1 Basic thoughts

Like the message protocol described in the last section, this transport protocol also consists of a well defined set fields. Therefore specific parameters, which are the same in all messages are always stored at the same position. Thus the same advantages apply to this protocol as listed in the message protocol section: parsing is easy and a lot of cpu time and memory is saved.

The strings, used by the connect message to transmit the credentials for the authentication, are encoded as described by the message protocol. A sequence of strings is initiated by a two byte field where the number of strings which will follow is encoded. The next two bytes contain the size of the string and afterwards the string itself follows including the final '0' byte. If more than one string should be encoded, they will be placed after the end of the previous string with their size in front of them. Also see the string description in the message protocol section for further specifications.

By using this protocol there will be situations where errors occur. Perhaps the connect message failed or the SSH connection to the server screwed up

and therefore the SSH server terminates the connection. Furthermore it is possible that the client wants to terminate the old and establish a new connection to the client daemon. In this case the kernel can't send a disconnect message, because this would cause the client daemon to exit. In all these cases the connection is shut down, which signalizes the other communication endpoint that this connection is terminated, but it is expected that a new connection will be established by the initiator of the previous connection. Thus no error messages and message identifiers are required. As this protocol is only used to communicate between kernel and client daemon, there will be no problems like a cut wire, where at least one communication endpoint doesn't notice that there is no connection any more. The communication is only handled by the kernel and doesn't leave the machine.

5.3.2 Header

To minimize the effect of an additional protocol, which is inserted between the underlying TCP/IP protocol and the message protocol, the header is reduced to only five bytes. This ensures a good header/payload ratio most of the time, because the size of the message protocol operations is at least 12 bytes and is often longer. The header consists of the size of the transmitted data and an opcode field, which specifies how this packet should be treated.

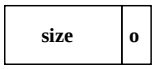

Figure 5.11: header

type	description	length in bytes
size	total size of the payload	4
o	command code	1

The maximum size of a packet must not exceed 4 GBytes. In a real environment, the size of a transmitted packet will probably range from a few

bytes to the buffer size of the client or server, which should be around 8 KBytes. Furthermore the maximum packet size isn't related to the maximum packet size of the message protocol, which is also 4 GByte. Packets from the message protocol can be split up into several chunks. These fragments are reassembled respectively concatenated before the data is passed to the message parsing engines. The size includes the payload but not the header itself, which is five bytes long in all cases. This value must be stored in network byte order also known as big endian format.

Because of the fact that there are only three types of messages defined by the protocol (connect, disconnect and data), the opcode field can be very short. One byte should be sufficient in this case.

Chapter 6

Implementation

This chapter contains the description of the second big part of the diplomathesis about the Secure Internet File System: the implementation of a prototype to prove the viability of the concept.

This will be the implementation of a server module for the OpenSSH server, the client daemon (sifsd) and a kernel filesystem driver for the Linux operating system. Due to the lack of time and the fact that I've invested a lot of time into the protocol design, the prototype will only be a read-only filesystem driver and server module. All the basic functionality is already implemented and it should be easy to exend the current work to get a full featured client and server. If necessary, I will point out some problems, which have to be solved on the way to a complete client and server.

6.1 OpenSSH Server Module

The server module for the OpenSSH framework is responsible for performing operations requested by the clients and returning the result of these operations. This may contain only the returned status of the performed operation or additionally include data transfers of file content or file metadata.

6.1.1 Description

The main function of the sifs module endlessly calls two functions in a loop until a fatal error occurs or the connection is closed by the other end. In this case the program terminates itself. The first function tries to receive a complete header, which is 12 bytes long. The second function is a dispatcher. It extracts the opcode encoded into the header by using macros which convert the values into host byte order and use the opcode as an index for a jump table. This table is programmed as a switch/case statement and created by the compiler automatically. In different cases statements the appropriate function is called, which perform the requested operation, like mounting a filesystem, opening a file or returning the file attributes. All these functions take only one parameter which is of the same type for all: a structure which consists of all necessary information needed by the functions. This includes the file descriptors for input and output, buffers for storing arrived and built packets, and the path of the directory which is mounted by the client. Furthermore, if such a function is called, the header read by the main loop is stored into the input buffer.

For a read-only file system, the messages listed below must be implemented by the server module. For the full list of all available messages and their description see Appendix A.

SIFS network messages
mount
unmount
statfs
getattr
readlink
readdir
open
release
read

All of them perform similar steps now summarized:
First of all the sequence number of the incoming packet is copied to the result buffer, so that the client can associate the reply to a former request. There are two buffers available, because the request payload can be used as input to a function writing directly to the output buffer. Thus it might be possible to corrupt the input of the function if only a single buffer would be used. Then they do some checks about the validity of the header, especially about the packet length: can the packet contain a payload and if yes, does it exceed the maximum allowed length for such a packet?

In addition, the context is examined. It is for example not valid to send a request to open a file before the filesystem isn't mounted. If the validity checks fail, the remaining packet is consumed (read from the socket), its content dropped and an error message (a packet, which contains only the header including the error number) generated and sent to the client. Only if the header has passed these checks, the remaining bytes are stored into the request buffer and the content is decoded. Before the parameters in the content are used to feed the functions performing the requested actions, they must also pass a few tests. Only then the needed operating system syscalls are executed with the appropriate parameters. If the parameters are valid and the system operation has completed successfully, the returned information is used to build the reply. If the system call failed for any reason, the error message returned by the function is mapped to one of the error classes known by the SIFS protocol and an error message is generated. After the reply packet (with success or failure messages) is built, it is sent to the client and the function returns to the main loop.

The SIFS protocol was designed to deal with the situation of a reconnect, where the server terminates the connection and exits. This is the reason why the protocol uses file names rather than server side file descriptors to reference files located on the server even on already opened files. As a consequence, the file offset must be transfered with each read or write message because no file pointer is maintained per client on the server side.

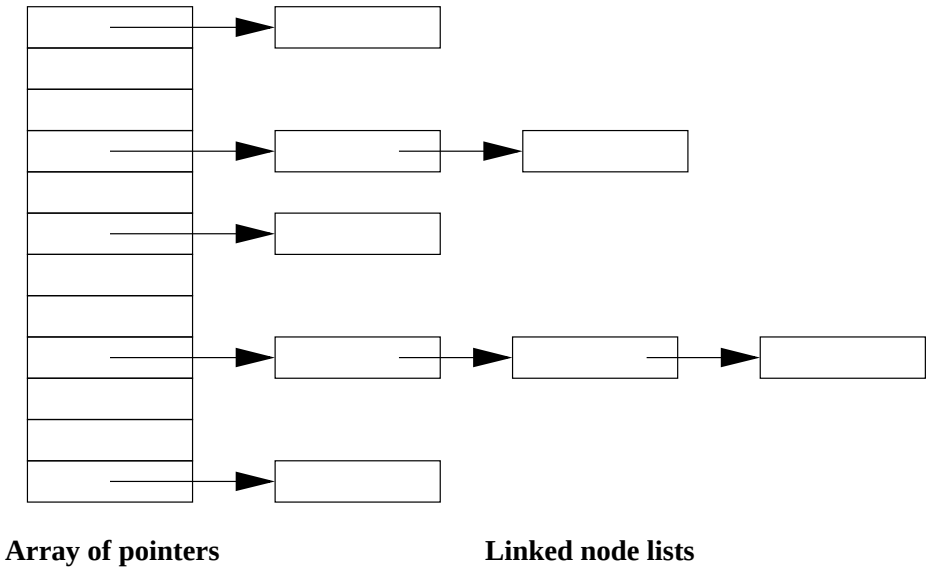

Figure 6.1: Open hash table

To access already opened files in a fast way without the availability of the file descriptor from the client, an "open hash table" (hash table without a limitation in the number of entries) was used. All characters of the file name and the path are XOR'ed and the resulting eight bits are stored in a variable. The content of this variable (the digest) is now used as an index for an array, which contains pointers to linked lists of table entries. Each node of these lists represent an open file with file name, file descriptor and usage counter. Therefore, a file must be opened only once on the server. There are supplementary functions for inserting, removing and retrieving an entry as well as increasing and decreasing the usage counter of the file. If the usage counter drops to zero, the file descriptor can be closed and the entry removed safely from the hash table. In the case of an error the sifs server modules exits and a reconnect from the client occurs. After that, all information about previously opened files are lost and the sifs module must be able to reopen files on read or write requests. For this reason, the sifs module also has to reply to close requests with a successful completion even if the file isn't opened any more.

6.1.2 Problem

One problem is currently not addressed by the implementation. If a locking mechanism should be implemented, there is a need for a program, which performs the locking operations for a user on the server. An external process is necessary because otherwise the sifs module would wait sending a reply until the lock is granted, and would therefore block all further operations on the client filesystem. The process must be able to send the result of a locking request asynchronously to the sifs module to indicate that the requested lock is now available. Thus, the main loop must be extended to check for such callbacks.

6.2 SIFS Client Daemon

Because of the need to keep encryption out of the kernel, there must be a daemon on the client side, which establishes an authenticated and encrypted tunnel to the server and forwards all messages between the client and the server back and forth.

6.2.1 Description

This daemon called sifsd or client daemon is set up by executing the sifs-client program. It must be executed by a user, who wants to mount a remote filesystem on his client to a directory of his choice which must be owned by the user. The parameters, which have to be provided by the user, are the server name or its IP address, the remote login name of the user and the port which should be used by the sifs-client program to listen for the connection from the kernel.

The sifs-client program creates a socket on the loopback device of the client machine with the given port. This port will then be used to listen for incoming connection requests from the kernel and the resulting connections are one end points for the messages encapsulated into the packet protocol.

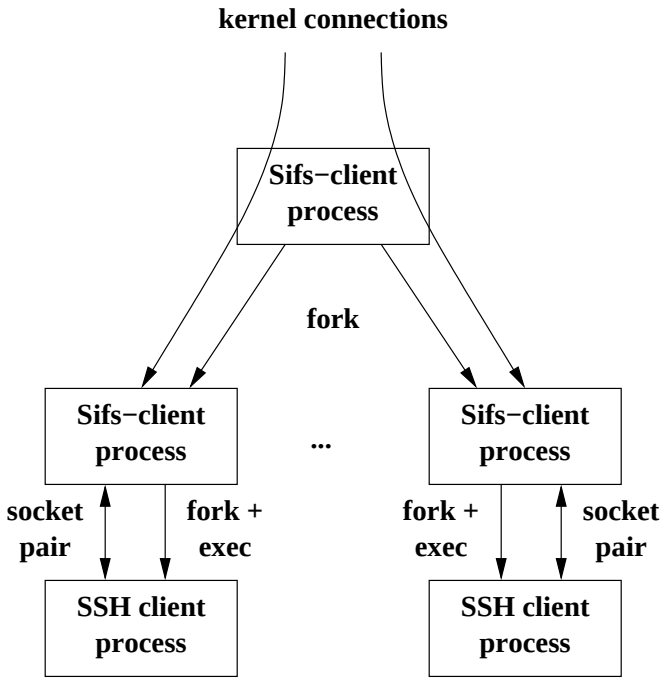

Figure 6.2: Forked child processes in the sifsd

If a connection request from the kernel filesystem driver arrives, the sifs-client program forks a child process. This process is responsible for handling all further data transfers to and from the kernel for this connection. Thus it also has to create the authenticated and encrypted tunnel to the server by spanwing a SSH client, which is connected to the child process by socket pairs. The SSH client on the client machine is executed with the server URL and the name of the user, who wants to authenticate himself against the server, as parameters. Furthermore, another few options for the SSH client are set, among them the option for the request to execute the sifs module on the server side. The SSH client will ask the user for his password and will transmit it to the server after establishing an encrypted tunnel. Now, the child process of the sifs-client program can enter the loop, which checks for the availability of new messages arriving from the kernel or the server. This is done by calling select on both socket file descriptors. Select will block the process until new data is available on one of these descriptors. If select returns, it has to be examined which descriptor is ready for reading and then

the message must be received and sent to the other end, either to the kernel or to the server.

The kernel can send a disconnect message specified by the packet protocol, which is the signal for the daemon to close the connection to the SSH client respectively server and terminate itself. After closing the connection to the SSH client, the client will handle the shutdown message sequence used by the SSH protocol and finally terminate itself. The other case is the termination of the connections by the server in case of an error detected by the sifs server module. Then the sifs-child process, which is responsible for this connection, must also close the remaining connection to the kernel and initialize itself for a reconnect from the kernel.

6.2.2 Problem

There seems to be a problem with the SSH client. Reconnecting to the server is only possible if the user is typing his password again into the prompt. An automatic input of the password from another program is currently not possible and seems to require patching the SSH client.

6.3 Kernel Filesystem Driver

The filesystem driver for the Linux kernel acts as a mediator between the Virtual Filesystem Switch (VFS) layer and the SIFS network messages. It presents in combination with the VFS layer a view of the remote files and directories to the applications running on the client.

6.3.1 Description

The SIFS filesystem driver is implemented as module, which can be inserted into the kernel at runtime. Thus, the module contains functions for registering and unregistering the filesystem implementation in the Virtual Filesystem Switch (VFS). They are executed by the macros *module_init* and *module_exit*

after inserting respectively removing the SIFS module from kernel space by the insmod and rmmod programs.

Furthermore, the driver implementation depends on the network functionality provided by the kernel like all other network filesystems (NFS, CIFS, etc). Unlike the others, I decided to place an abstraction layer on top of the socket implementation of the Linux kernel. This layer provides an interface, which is similar to the system calls used by user space applications, and this layer should make it easier to port the driver to other operating systems.

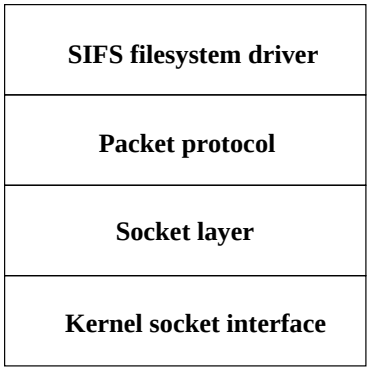

Figure 6.3: Communication layer between kernel and SIFS driver

On top of the abstraction layer the packet protocol is implemented, which is used for the communication to the client daemon. The packet protocol provides functionality for creating and closing sockets as well as connect, disconnect, read and write operations. Some of these operations, like create or close, are almost directly mapped to the functions of the abstraction layer, but the rest does a little bit more. After a successful packet_connect operation for example, it can be assumed that the client daemon already has established an authenticated and encrypted tunnel and all transmissions by *packet_read* and *packet_write* are safe. Furthermore the *packet_disconnect* operation causes the client daemon to terminate itself, whereas *packet_close* only terminates the connection. In case of *packet_close* without *packet_disconnect*, the client daemon is waiting for a reconnect.

After describing the dependencies to the other parts of kernel (excluding the description of the VFS, which was already explained in chapter four), it is possible to explain the implementation of the filesystem driver in detail. It should be obvious that the SIFS network messages (see Appendix A) must be more or less directly mapped to one or more VFS interfaces. There are the following interfaces and messages, which have to be implemented for a read-only filesystem:

Linux VFS interface	SIFS network message
read_super	mount
put_super	unmount
statfs	statfs
getattr	getattr
lookup	getattr
revalidate	getattr
readlink	readlink
follow_link	readlink
readdir	readdir
open	open
release	release
readpage	read
permission	locally implemented
iget	locally implemented

The functions, which implement the VFS interface are translating the data returned by the message replies into a format the VFS layer is waiting for. They either call separate functions, which build the requests and return the results of the reply or implement them themselves.

The sequence of steps to send a request and process the reply are the same among all functions. At first, they must obtain some data related to this connection. Especially important is the socket for communication to the server and the current sequence number for the next request. This information is

stored in a structure attached to the superblock, which provides a generic pointer, where these private data can be placed into by the *read_super* function. Because of the fact that applications can access files in parallel, not only on multiprocessor machines, direct or indirect read and write operations on the members of the private structure must be protected by a semaphore. This semaphore is also a member of the structure. Each request for an operation begins by getting the new sequence number from the private data structure atomically. Only then the header of the request can be set up and additional parameters can be encoded into the buffer as payload. After the request packet is completely built, it can be sent to the server. While the packet is sent and until the reply is completely received the semaphore must be hold. This ensures that the requests or replies are not mixed with other ones, which would lead to corrupt data, but also leads to two problems described in the next section. The reply should now be stored into the buffer and can be further examined. In case of a failed operation the error codes have to be translated to codes known by the kernel. Otherwise, if the reply contains a payload, it has to be decoded and returned to the calling function in a format which is understandable by the caller.

Now, after the basic procedure should be clear in principle, each function implementing VFS interface can be explained in detail. Because of the naming policy in the Linux kernel, the function names are prefixed by the name of the filesystem, which is "sifs_" in our case.

The first one is *sifs_read_super*, the function used as entry point to mount the remote filesystem on the client. It must be declared as entry point by a special macro (DECLARE_FSTYPE) and is responsible for creating a connection to the server over the client daemon by using the packet protocol. Thereby it has to parse the options provided by the *mount* syscall, which contains information about the listening port of the client daemon, the username and the password for authentication. With this information, it can establish an authenticated and encrypted tunnel by using the client daemon and send a mount request to the server by calling *sifs_mount*. Furthermore, it has to

request information about the mounted filesystem, which is provided by an *sifs_statfs* reply as well as information about the mounted server directory provided by *sifs_getattr*. This information has to be filled into the newly created root inode respectively the superblock, where also the allocated private structure containing the connection data, the uid/gid numbers for mapping and the initialized semaphore, is attached. On a successful operation, the filled superblock is returned.

Sifs_put_super is the reverse operation to *sifs_read_super*. It is responsible for unmounting the filesystem and closing the connection after sending the client daemon the signal to terminate itself (*packet_disconnect*). Furthermore it has to free the allocated private structure pointed to by the superblock. The created root inode is freed by the VFS layer if it isn't used any more.

The implementation of *sifs_statfs* can be directly mapped to the SIFS statfs message. It is called by *sifs_read_super* and if a user wants some information about the remote filesystem by calling the *statfs* system call. The purpose of *sifs_statfs* is to return the total number of blocks, free and available blocks as well as the total number of files and available files (inodes). Moreover, it provides information about the maximum file name length, the preferred block size and the magic number of the superblock. The preferred block size is set by *sifs_statfs* to 8 KB so it does match the optimum found for the NFS protocol, which is very similar. Moreover the magic number is also set by *sifs_statfs*, which distinguishes slightly different filesystem implementations of the same type. Originally the magic numbers were established by Minix, the operating system created by Andrew Tannenbaum.

Lookup returns the attributes for a given file name, filled into an inode created by *sifs_iget* (described later). It also adds the file name and the corresponding inode to the dentry cache. If the lookup failed, a negative dentry with no inode pointer is added. Like *revalidate*, it calls *getattr* directly, because up to now no cache is maintained by the driver. The *getattr* reply returns various information about the requested file, like the mode bits, the number

of links, uid and gid numbers, time information (create, access and modify timestamp) and the size of the file. *Getattr* is also responsible for the mapping of the uid and gid numbers between the client and the server. This ensures that the files on the server, which belong to the authenticated user can only be accessed by the user and his primary group on the client. All other files on the server appearing locally are mapped to the root account, but are not accessible by the super user, because the server module only has access to the files of the authenticated user.

The *readlink* and *follow_link* functionality is implemented into the VFS layer. They only need the link content provided by the SIFS readlink message, which is implemented by *sifs_getlink*.

Readdir is a little bit more complex. After the SIFS readdir reply is received by *sifs_readdir*, the encoded file or directory name strings must be translated into an appropriate format for the VFS layer. The VFS layer provides a pointer to a function, which accepts one string entity and a few more parameters for storing the names into the dentry cache and providing them to user space applications on request. The cache must be filled with the names starting at a given position until all names are inserted. The type of inodes related to the names filled in is normally not known. Therefore, the VFS layer will ask *lookup* for this later.

The *open* and *release* implementations are obvious. They only return the successful completion of an operation or an appropriate error message to the caller and are not responsible for any file descriptors. Instead the whole work is done by the server and the VFS layer.

The last category of functions related to SIFS network messages are the address space operations. Some of them are implemented by the VFS layer, but for the read-only filesystem the implementation of *sifs_readpage* is necessary. *Sifs_readpage* calls subsequently *sifs_read*, which implements the SIFS read request, until a complete memory page (usually 4 KB) is filled. Then

the page flags are updated and all remaining errors cleared. The page content representing a portion of a file is now ready for usage by applications.

For the *permission* function an equivalent SIFS message doesn't exist. It is responsible for examining the mode bits of a file and granting or denying access to an application. Because the SIFS doesn't implement extended features based on rules - like Access Control Lists (ACLs) for example - which allows or denies specific operations (append, delete or others) yet, it can rely on the VFS implementation of permission. Especially after the uid/gid mapping done by *getattr*, it can be assumed that only authorized users and groups can access the remote files, because then the traditional Unix semantics is trustworthy.

A special case is also the implementation of *sifs_iget*, which is usually implemented by the VFS layer. Traditional Unices and therefore also the Linux VFS layer use the term "inode" for a data structure, which contains all information about a file or directory on a filesystem. The different instances of this data structure represent the files or directories located on the filesystem and are identified by a unique inode number (unique on the filesystem). On filesystems, which don't use inode numbers like Microsoft's FAT filesystem, or filesystems, whose inode numbers are not unique on the local machine as in network filesystems, unique numbers must be created by the kernel. The root inode of a filesystem is the only exception: it always uses a fixed number, e.g. "1" or "2". *Sifs_iget* is thus responsible for creating a new inode structure and filling it with the appropriate values. Furthermore, it has to examine the type of inode (file, directory or symlink) and store the appropriate structures holding the specific function pointers in the inode.

6.3.2 Problem

In the current implementation of the read-only filesystem the functions called by the VFS layer, lock the socket for the server connection by setting a semaphore before sending a request and releasing the semaphore after re-

ceiving the complete reply. In fact, they keep the socket locked until the transaction is complete. Thus, it is currently not possible to implement the *sifs_lock* function, which requires asynchronous replies until the locking scheme is more fine grained. The server normally wouldn't send a reply in case of a lock request until the lock is granted, except the lock request was a non-blocking request. Therefore, all further operations on the mounted filesystem would block until the lock is granted. This means that the filesystem is unusable in the meantime. In addition, locking the socket until the reply arrives decreases the performance drastically if several applications perform operations on the filesystem provided by the SIFS kernel driver. They have to wait until a transaction is complete, independent of the time it will last and can not perform operations in parallel.

The solution would be the implementation of a read and a write semaphore instead of a single one. Sending requests and receiving replies can then be done independently by more than one filesystem operation. Furthermore, it is necessary to implement a dispatcher for all replies from the server, which must be able to receive the reply packets and distribute them to the waiting filesystem functions based on their sequence number. One possibility would be a solution based on semaphores.

Chapter 7

Test

Finally, the implementation have to be tested and remaining bugs must be fixed. This chapter contains a description of the testing environment as well as the description of the tests of the modules. At the end, after all module tests are done, there is also an integration test including all components using real world data, where the whole implementation must prove its viability.

7.1 Environment

The development was done on a single machine, which was also be used as a testing bed for the server module, the kernel filesystem driver and the client daemon (sifsd). The installed Linux distribution on this machine was Redhat 7.2 which was updated to the latest patches, including a new kernel version. Thus, the following version were used for development and testing:

- kernel-2.4.9-13
- openssh-2.9p2-12
- gcc-2.96-98

All source code related to the kernel filesystem driver was compiled as module, which can be inserted into the kernel at runtime, and with default flags used by the kernel make file.

Finally, for the final integration test over the network, a second machine was needed. This computer contained an image of a Redhat 7.0 installation without any patches and was only used for the network test. Thus, the only interesting software package on this machine is the OpenSSH server package, whose version was 2.1.1p4-1. Both machines, the computer used for development and the second machine for the network test were built on the Intel x86 architecture.

7.2 Procedure

The test only consisted of a functionality test, proving the correctness of the software implementation. It didn't contain a performance test, which benchmarks the execution speed of the operations.

To understand the steps performed by the implemented software, no memory debugger was used. Instead, various profiling information was inserted into the source code. This profiling code consists of macros, which print the current action and interesting parameters to the screen. The profiling output can be enabled by defining SIFS_PROFILE at compile time, otherwise the compiler generates a binary without any profiling data. Moreover there are error messages displayed if something failed, but they can be also suppressed by erasing the "SIFS_DEBUG" macro.

Both component tests, the test of the server module and the test of the kernel filesystem driver, were done by feeding the modules with input packets and examining the resulting output. For each type of network message described in the appendix and implemented by the server and kernel module, there was a generated test packet as input. Therefore, test programs for automating this process were written, sending valid packets to the modules and displaying the output in an human readable form. After the remaining bugs were fixed and the output was equivalent to the expected one, the test programs were modified to generate packets with errors to examine the

behavior of the server and kernel module. The errors were introduced by hand in random fields of the packets, but not all error cases are tested in all functions. Then, after fixing errornous behavior in the tested code, the packet fields were reset to correct values and the test was executed a last time. All these tests were performed on a single machine with the usage of the loopback network device, thus no second computer was needed up to now.

For the final integration test an environment containing real data was used, in this case directories created by the Linux distribution on the computer where the software was developed. First of all the low level parts of the client daemon, like the packet protocol and its interface to the remaining client daemon code, were tested in conjunction with the kernel module. This test was extended successively by adding the server module without the usage of the OpenSSH until the client daemon forwards all requests and replies based on real data without errors. Finally the direct connection from the client daemon and the server module was replaced by an OpenSSH connection. This connection was established over an ethernet network from the development computer to the server machine, where Redhat 7.0 was installed.

7.3 Conclusion

After finishing the tests, the implemented software is of good beta quality and is ready for testing in a wider public. There are only two problems currently not fixed: the number of links to a file or directory is not shown correctly and the OpenSSH server seems to stop forwarding replies to the client after a certain amount of bytes.

The first problem, the number of links is related to the VFS/driver interaction. The VFS interface doesn't provide the possibility to return the number of links by a call to *getattr*. Thus, their number is known after a *getattr* request by the filesystem driver, but it can not tell this information the VFS directly. It seems to me that this is handled in another way currently not known by me.

The other problem regarding the OpenSSH server is much more important. By performing actions on the client machine, which requires several requests and replies (around 23 or 24), the server stops forwarding the next reply to the client machine. Up to now I don't know why, but I'm sure the problem is either due to a mistake of myself in the code of the sifs-client or a bug in the OpenSSH software. If the client daemon creates the server module locally without OpenSSH client and server, all requests and replies are forwarded correctly without limitations.

Chapter 8

Perspective

Up to now there is only a prototype for a read-only filesystem implemented. The whole functionality for creating, deleting or writing into files or directories is currently missing respectively there are only the function stubs available. The implementation of these functions on the client and the server side can be done rather quickly within a few weeks, but the real problem is the implementation of the *lock* call, which is very different from other calls, because of its asynchronous nature. The lock message requires the kernel driver to send a request to the server without blocking other requests while waiting on the reply. Therefore the implementation of the dispatcher for the replies from the server mentioned at the end of the implementation chapter is absolutely necessary, but requires a lot of additional thoughts The server module also needs some special code in form of a separate process handling these lock requests and it will cost some time implementing it too.

Furthermore the current implementation is not optimized for speed and contains no caches. These caches, when they are implemented, could be used to save recently requested file attributes for speeding up the *revalidate* call or to store the result of a *readdir* call for a certain amount of time. There may be a wide field for optimizations and good references are the implementations of the NFS, SMB and NCP filesystem driver in the Linux kernel.

The last thing required would be the patching of the OpenSSH client, because it doesn't accept the password on stdin yet. This leads to problems when an error occurs and the kernel filesystem driver has to reconnect to the server. In this case, the SSH client requests the user to type in his password on the terminal again instead of taking the password as input from the parent process (the client daemon).

Bibliography

[1] http://www.webopedia.com/TERM/C/CIFS.html

[2] http://whatis.techtarget.com/definition/0,289893,sid9_gci549024,00.html

[3] http://www.sans.org/infosecFAQ/threats/middle.htm

[4] http://searchSecurity.techtarget.com/sDefinition/0,,sid14_gci557336,00.html

[5] http://www.heise.de/tp/english/inhalt/te/6929/1.html

[6] http://www.heise.de/tp/deutsch/special/enfo/7220/1.html

[7] http://www.inch.com/ agagescu/hp/Ntpass.htm#pweqv

[8] http://www.faqs.org/rfcs/rfc1813.html

[9] http://rr.sans.org/unix/nfs_security.php

[10] http://www.cerias.purdue.edu/coast/satan-html/tutorials/vulnerability_tutorials.html

[11] http://www.microsoft.com

[12] http://rr.sans.org/win/SMB.php

[13] http://www.novell.com

[14] http://security.tsu.ru/info/nw/ncp.txt

[15] http://www.coda.cs.cmu.edu/

[16] K. Washburn, J. Evans: TCP/IP Running a Successful Network, Second Edition, Addison-Wesley, 1996

[17] http://www.distributed.net/

[18] http://csrc.nist.gov/encryption/aes/

[19] http://www.ssh.com/

[20] http://www.openssh.com/

[21] B. Goodheart, J. Cox: The magic garden explained - the internals of Unix System V R4, Prentice Hall, 1994

[22] http://www.kernel.org/

[23] Jeff King: peff@fenris.cc

Appendix A

SIFS message protocol description

A.1 Filesystem related operations

There are only three filesystem related operations: mounting, unmounting and stating a filesystem. The mount operation connects a part of the server filesystem tree to the local tree on the client. Unmount is the opposite operation, because it removes mount points previously installed by the mount operation. Without them the client would not be able to perform further operations on files and directories which reside on the server or make a clean disconnect at the end. Thus, file or directory operations will always fail without a previously sent mount command and all requests sent after an unmount are also invalid. Finally, statfs returns information about the filesystem.

A.1.1 Mount

Description:

The client requests for allowance to access a directory and its subdirectories on the server by using the specified version of the protocol. Depending on the access rights of the user on the server or an unsupported protocol version, it may be denied by the server. Furthermore, only one mount operation per

connection is allowed and therefore only one directory per child process on the server can be exported. Thereby a new connection must be established for each new mount operation.

Request:

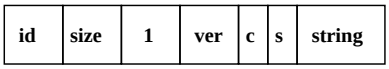

Figure A.1: mount request

type	description	length in bytes
ver	protocol version	4

The protocol version specifies which version the client is able to speak. If this version is not supported by the server, an error will be returned and the client may decrease the version number and retransmit the request if supported. The string contains the path which is used as the new working directory for all further requests.

Reply:

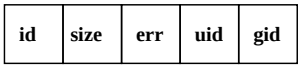

Figure A.2: mount reply

type	description	length in bytes
uid	user id	4
gid	primary group id	4

The returned numbers are the user and group ID of the user who logged into the server and are used to map the IDs between the client and the server.

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component is not a directory
SIFS_EACCES	Mounting is denied due to missing rights
SIFS_ENOTSUP	Protocol version is not supported
SIFS_ESERVER	Internal server error

On success, zero is returned in the error field, otherwise it is set to one of the listed error numbers.

A.1.2 Unmount

Description:

Removes a mount point on the server which ist then not accessible by the client any more. This call is used to initiate a clean shutdown. If this call is successful, the connection must be terminated.

Request:

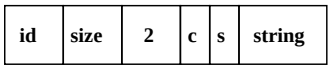

Figure A.3: unmount request

The string contains the location of the directory, which is mounted by the client. The server is requested to no longer export the mount point to this client and is advised to free its resources related to the mount point.

Reply:

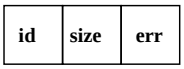

Figure A.4: unmount reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	Directory wasn't mounted previously
SIFS_ESERVER	Internal server error

On success, zero is returned in the error field, otherwise it is set to one of the listed error numbers.

A.1.3 Statfs

Description:

Returns various information about the mounted filesystem.

Request:

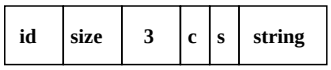

Figure A.5: statfs request

This request contains a path to an existing file on the server. The filesystem where this file resides will be queried for various information and these values are returned.

Reply:

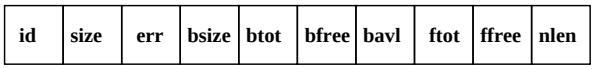

Figure A.6: statfs reply

type	description	length in bytes
bsize	preferred block size	4
btot	total number of blocks	4
bfree	number of free blocks	4
bavl	number of available blocks	4
ftot	total number of files	4
ffree	number of free files	4
nlen	max. file name length	4

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EACCES	Search permission is denied for this path
SIFS_EIO	An I/O error occurred
SIFS_ENOTSUP	Operation not supported
SIFS_ESERVER	Internal server error

On success "statfs" returns zero, otherwise one of the error numbers listed above is returned. Furthermore "statfs" returns various information about the blocks and files, which are available and free. The available number of blocks may be less than the total number of blocks, because Unix filesystems normally preserve a small amount of space, which is only available for the administrator. Moreover the maximum length for file names is included in this reply.

A.2 Inode related operations

The word "inode" is as a generic term for both file and directory objects. This term is often used in documents related to the Unix virtual filesystem (VFS) and I will continue to use it here.

All operations described below can be performed by specifying the location of an inode which is described by its name. Even those requests, which are stateful like read or write do not need to pass a file descriptor. Instead the file descriptors are stored on the server side and only the string which contains the location of the file or directory is transferred. This eases error handling a lot. In case of a recovery from a disconnect, there are no invalid file descriptors from the server stored on the client side.

A.2.1 Open

Description:

"Open" ensures that the file described by the given file name exists and is accessible by the user who wants to perform subsequent operations on this file. This call also compares the requested rights (read, write or both) with the file permissions attached to this file. Despite the POSIX open syscall this operation doesn't return a file descriptor which is normally used as a reference for further access to files. Instead all operations will use the file name for subsequent access. This is because a file name will not become invalid in case of a disconnect/reconnect caused by an error.

Request:

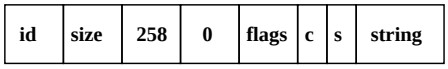

Figure A.7: open request

type	description	length in bytes
0	32 bit representation of zero for padding	4
flags	open mode	4

The string points to the file (regular or special file) which should be opened. The flags specifies the access mode which is used to open the file. Possible values are 0 for read access only, 1 for write access only and 2 for both read and write access. Furthermore these bits can be or'ed with the octal values 04000 for nonblocking access (return an error, if open or subsequent operations would block) and 010000 for synchronous file operations (call will block until all data is written to the underlying medium).

Reply:

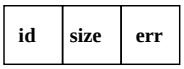

Figure A.8: open reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	The file doesn't exist
SIFS_ETEMP	File access is currently not possible
SIFS_EACCES	File access is denied due to missing rights
SIFS_ESERVER	Internal server error

On success "open" returns zero, otherwise one of the error numbers listed above is returned.

A.2.2 Create

Description:

Creates a new entry for a regular file on the filesystem. Unlike Unix creat(), it doesn't return a file descriptor on success. Furthermore, only regular text or binary files can be created by this call. If special files like pipes should be created, use mknod instead.

Request:

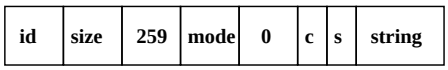

Figure A.9: create request

type	description	length in bytes
mode	the posix file mode attributes	4
0	32 bit representation of zero	4

Parameters are the create mode, a zero value for padding and a string. The mode argument specifies the permissions which are granted in case of further access, but may be altered by the server. The three least significant bits (000X) are used to describe the access rights of all users, the next three bits (00X0) to describe the rights of the users which are in the same group as the creator and the third three bits (0X00) to describe the access rights, which are granted to the creator himself. The access rights are split into the read, write and execute permission. Read is represented by the octal value of four, write by two and execute by one. They may be or'ed (e.g. 04 and 02 = 06) to get read/write access or any other combinations of access rights. All other remaining bits must be set to zero. The described mode is the same as specified by the POSIX standard and used by all Unix implementations.

Reply:

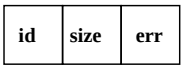

Figure A.10: create reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EEXIST	File already exists
SIFS_EPERM	File creation is denied permanently
SIFS_EACCES	File creation is denied due to missing rights
SIFS_ENOSPC	There was no space for a new file available
SIFS_ESERVER	Internal server error

On success, zero is returned in the error field, otherwise it is set to one of the listed error numbers.

A.2.3 Mkdir

Description:

Creates a new directory inode on the filesystem at the specified location.

Request:

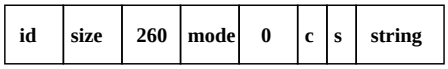

Figure A.11: mkdir request

type	description	length in bytes
mode	the POSIX file mode attributes	4
0	32 bit representation of zero	4

Like "create" for regular files, the mkdir request contains the mode of the newly created directory, a zero value for padding and the string representation for the path. For a description of the mode bits look at the description of the create request. Instead of execute permission, user can get the right to search (traverse) the tree. The submitted mode may be altered by the server.

Reply:

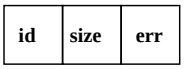

Figure A.12: mkdir reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EEXIST	Directory already exists
SIFS_EPERM	Creation is denied permanently
SIFS_EACCES	Creation is denied due to missing rights
SIFS_ENOSPC	There was no inode for the directory available
SIFS_ESERVER	Internal server error

The reply of the server will be zero on success or one of the error numbers listed in the above table on failure.

A.2.4 Mknod

Description:

Creates an inode, which may be a device special file or a named pipe. This request is optional, because it is Unix specific and not described by a POSIX standard.

Request:

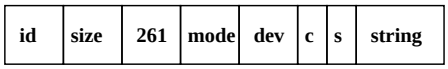

Figure A.13: mknod request

type	description	length in bytes
mode	the device type and posix file mode attributes	4
dev	major and minor number of the device	4

The access rights, which are part of the "mode" field are the same as described by the create request. Additionally the type of the newly created special file is encoded into the mode as well. Possible octal values are 010000 for a named pipe (fifo), 020000 for a character special file and 060000 for a block special file. These bit masks have to be or'ed with the masks representing the access rights. Also the access rights may be modified by the server.

The field "dev" contains the major and minor number of the new device if a character or block special file should be created. Otherwise it is ignored.

Reply:

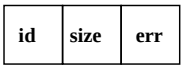

Figure A.14: mknod reply

error number	description
SIFS_EINVAL	Mode argument was invalid
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path doesn't exist
SIFS_EEXIST	File already exists
SIFS_EPERM	Operation is denied permanently
SIFS_EACCES	Creation is denied due to missing rights
SIFS_ENOSPC	There was no inode available for a new file
SIFS_ENOTSUP	Operation is not supported
SIFS_ESERVER	Internal server error

The reply to a mknod call contains zero on success or one of the above error numbers on failure.

A.2.5 Release

Description:

All files opened previously must be closed. This operation acts as a signal to the server that the reference counter for the file can be decremented, and if the counter reaches zero, the file can be finally closed.

Request:

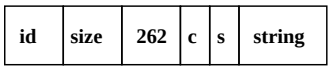

Figure A.15: release request

The only parameter for a "release" request is a file name of a previously opened file.

Reply:

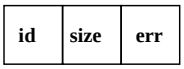

Figure A.16: release reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	The file was not opened previously
SIFS_EIO	An I/O error occurred
SIFS_ESERVER	Internal server error

On success "release" returns zero, otherwise one of the error numbers listed above is returned.

A.2.6 Unlink

Description:

Deletes a file name from the filesystem. If hard links are supported and this name isn't the last one referring to the specified file, then all used blocks on the medium arn't freed. Otherwise, the used blocks are freed immediately.

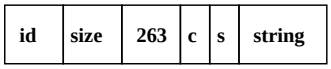

Figure A.17: unlink request

The unlink request is rather simple. It only contains a string, which is used to reference the file which should be deleted.

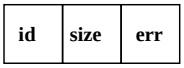

Figure A.18: unlink reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EPERM	Deleting a file is denied permanently
SIFS_ETEMP	File is currently used by another process
SIFS_EACCES	Deleting a file is denied due to missing rights
SIFS_EIO	An I/O error occurred
SIFS_ESERVER	Internal server error

On success zero is returned otherwise one of the error numbers listed above.

A.2.7 Rmdir

Description:

Deletes a directory name on the filesystem. The directory must be empty for a successful reply.

Request:

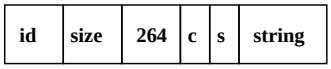

Figure A.19: rmdir request

Like the unlink request, "rmdir" is pretty simple. The only argument is the string which points to the directory which should be removed.

Reply:

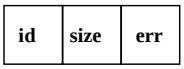

Figure A.20: rmdir reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EPERM	Operation is denied permanently
SIFS_ETEMP	Directory is currently in use
SIFS_EACCES	Operation denied due to missing rights
SIFS_ENOTEMPTY	Directory is not empty
SIFS_ESERVER	Internal server error

On success "rmdir" returns zero, otherwise one of the error numbers listed above is returned.

A.2.8 Link

Description:

"Link" creates a new name entry in the filesystem for an existing file. This is also known as creating a "hard link" to an existing file. Note that hard links - unlike symbolic links - can not span filesystems.

Request:

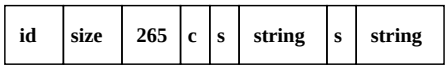

Figure A.21: link request

The link request consists of two strings: the path to the existing file and the path of the new link. Both can be relative or absolute paths. If the new link name already exits, it will not be overwritten.

Reply:

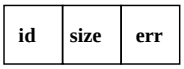

Figure A.22: link reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EPERM	Creating a link is not supported
SIFS_EACCES	Creating the link is denied due to missing rights
SIFS_EEXIST	File already exists
SIFS_ENOSPC	There was no space for a new file available
SIFS_EIO	An I/O error occurred
SIFS_ESERVER	Internal server error

On success "link" returns zero, otherwise one of the error numbers listed above is returned.

A.2.9 Symlink

Description:

Creates a new filesystem entry, whose content points to another file or directory. This symlink may point to a nonexistent file or directory and is in this case named a dangling link. Unlike hard links, symlinks are interpreted at run time by the operating system. The permissions of a link are checked if

the link itself should be modified (renamed, deleted or the content modified). They are not checked if the file or directory where the link points to should be accessed. Instead, the permissions of the referenced file are used.

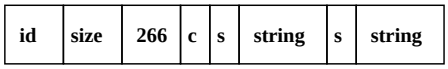

Figure A.23: symlink request

The first string contains the path where the symlink points to, which is also known as the referenced file. The second string is the name (and path) of the symlink file, which will contain the first string.

Reply:

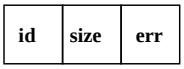

Figure A.24: symlink reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EPERM	Creating a symlink is not supported
SIFS_EACCES	Creation denied due to missing rights
SIFS_EEXIST	File already exists
SIFS_ENOSPC	There was no inode for a new symlink available
SIFS_EIO	An I/O error occurred
SIFS_ESERVER	Internal server error

On success "symlink" returns zero, otherwise one of the error numbers listed above is returned.

A.2.10 Readlink

Description:

Returns the contents of the given symbolic link. This request is optional because it is not described by a POSIX document.

Request:

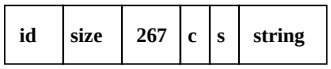

Figure A.25: readlink request

The string contains the path of the symlink file, which points to the original file.

Reply:

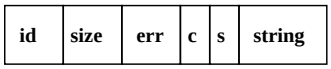

Figure A.26: readlink reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EINVAL	Not a symbolic link
SIFS_EACCES	Operation is denied due to missing rights
SIFS_EIO	An I/O error occurred
SIFS_ESERVER	Internal server error

On success "readlink" returns zero, otherwise one of the error numbers listed above is returned. If the request was successful the reply contains the content of the symlink (the path to the referenced file).

A.2.11 Rename

Description:

Renames an existing file or directory and moves it between directories if required. If the file which should be renamed is a symlink then the symlink is renamed, not the referenced file.

Request:

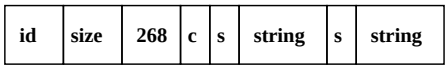

Figure A.27: rename request

The first string contains the path of the old (existing) file name, the second one of the new path.

Reply:

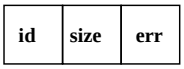

Figure A.28: rename reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EPERM	Operation is denied permanently
SIFS_ETEMP	Operation is currently not possible
SIFS_EACCES	Renaming is denied due to missing rights
SIFS_EINVAL	Invalid argument
SIFS_ENOSPC	There was no inode available
SIFS_ESERVER	Internal server error

On success "rename" returns zero, otherwise one of the error numbers listed above is returned.

A.2.12 Truncate

Description:

Truncates a file to the given size. All data, which was stored behind this size is lost after this call. If the file was smaller than the given size, the result is operating system dependent. Either the file is left unchanged, or the file is extended to this size and filled up with zero bytes.

Request:

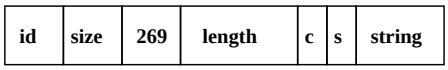

Figure A.29: truncate request

type	description	length in bytes
length	new length of the file	8

As parameter the length of the new file and the file name is put into the request.

Reply:

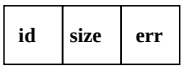

Figure A.30: truncate reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EPERM	Operation is denied permanently
SIFS_ETEMP	Operation is currently not allowed
SIFS_EACCES	Truncation is denied due to missing rights
SIFS_EIO	An I/O error occurred
SIFS_ESERVER	Internal server error

On success "truncate" returns zero, otherwise one of the error numbers listed above is returned.

A.2.13 Getattr

Description:

Returns the attributes of the specified file, which are also known as the meta data of a file.

Request:

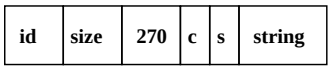

Figure A.31: getattr request

The string parameter specifies the file whose attributes should be returned.

Reply:

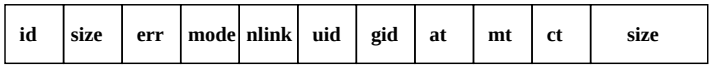

Figure A.32: getattr reply

type	description	length in bytes
mode	POSIX file access mode	4
nlink	number of hard links	4
uid	user id number	4
gid	group id number	4
at	last access time	4
mt	last modification time	4
ct	creation time	4
size	size of the file in byte	8

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EACCES	Access is denied due to missing rights
SIFS_ESERVER	Internal server error

On success "getattr" returns zero, otherwise one of the error numbers listed above is returned. Furthermore various attributes about the file are sent back by this reply. The possible access rights are described in the section about the "create" request. Additionally this field is or'ed with the type of that file. All possible bit masks for the known file types are described in the table below. The second field in the reply payload contains the number of hard links to a file respectively the number of subdirectories, if the given path referes to a directory. Also the user and group id numbers of this file are returned. These are the raw numbers and will probably differ from these known by the client or mapped to the false user. Therefore it is the job of the client to modify them as needed before they are used for granting or denying further access. Access to the files on the server is only granted if the rights of the user who mounted the filesystem allow this. The three time fields contain the number of seconds for the creation, last access and last modification since the Epoch (00:00:00 UTC 1970-01-01). The last field represents the current file size in bytes.

bit mask (octal)	description
0140000	socket
0120000	symbolic link
0100000	regular file
0060000	block device
0040000	directory
0020000	character device
0010000	fifo

A.2.14 Setattr

Description:

Sets new attributes for the given file. This call can change all attributes returned by the getattr request without the number of hard links and the size of a file. This can only be done by creating or removing subdirectories respectively invoking a truncate call or appending data at the end of the file.

Request:

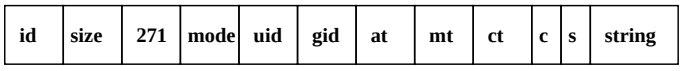

Figure A.33: setattr request

type	description	length in bytes
mode	POSIX file access mode	4
uid	user id number	4
gid	group id number	4
at	access time	4
mt	modification time	4
ct	creation time	4

For a description of the file access modes see the "getattr" section. The uid and gid can only be changed, if root was the user who mounted the filessystem. Furthermore the access and modification time can be changed, but the creation time is set by the operating system and normally can't be updated later by the user. In this case the value should be ignored.

Reply:

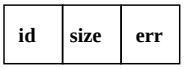

Figure A.34: setattr reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EACCES	Access is denied due to missing rights
SIFS_ESERVER	Internal server error

On success "setattr" returns zero, otherwise one of the error numbers listed above is returned.

A.2.15 Readdir

Description:

Returns the names of all entries located underneath the given directory.

Request:

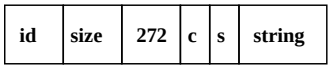

Figure A.35: readdir request

The "readdir" request contains only a directory name used to get all entry names underneath.

Reply:

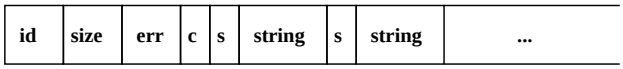

Figure A.36: readdir reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	A path component doesn't exist
SIFS_EACCES	Access is denied due to missing rights
SIFS_ESERVER	Internal server error

On success "readdir" returns zero, otherwise one of the error numbers listed above is returned. On success, one or more strings containing the names of the entries located underneath the requested directory are returned.

A.2.16 Read

Description:

Reads some data from the given file. This request is only valid if the same file was previously opened by the "open" request.

Request:

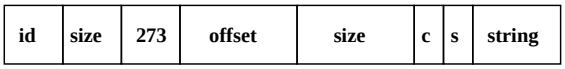

Figure A.37: read request

type	description	length in bytes
offset	position where to begin	8
size	number of bytes to read	8

To read a number of bytes from a file, an offset must be given which describes the position of the first requested byte and the number of bytes (size) which should be read.

Reply:

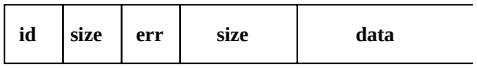

Figure A.38: read reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	The file was not opened previously
SIFS_EIO	An I/O error occurred
SIFS_EINVAL	Invalid argument
SIFS_ESERVER	Internal server error

On success "read" returns zero, otherwise one of the error numbers listed above is returned. Furthermore the requested data and their size is returned. This size may differ from the size initially sent, if there is less data available or the buffer on the server side was not big enough.

A.2.17 Write

Description:

Write the given data to the file on the server. This request is only valid if the file was previously opened by the "open" request.

Request:

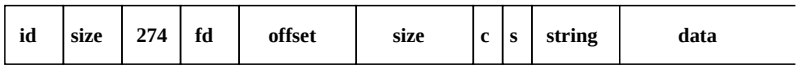

Figure A.39: write request

type	description	length in bytes
offset	position where to begin	8
size	size of the data	8
data	new file content	n

The offset specifies where to begin writing the data and size must be equivalent to the length of the given data.

Reply:

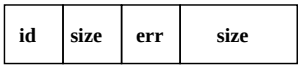

Figure A.40: write reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	The file was not opened previously
SIFS_EIO	An I/O error occurred
SIFS_EPERM	Operation is denied permanently
SIFS_EINVAL	Invalid argument
SIFS_ENOSPC	There was no more space available
SIFS_ESERVER	Internal server error

On success "write" returns zero, otherwise one of the error numbers listed above is returned. Furthermore the size of the data written to the storage medium is returned. This size may differ from the size initially sent, if there was less space available. If the data is written only partly, the operation returns no error.

A.2.18 Fsync

Description:

Writes all data related to this file (meta data and payload), which are cached by the operating system to the stable storage medium. This request returns after this operation is completed and is only valid on previously opened files

Request:

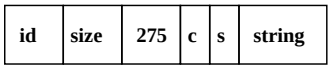

Figure A.41: fsync request

The string points to the open file whose content should be flushed to the stable storage medium.

Reply:

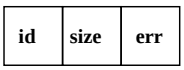

Figure A.42: fsync reply

error number	description
SIFS_ENAMETOOLONG	Pathname "string" was too long
SIFS_ENOENT	The file was not opened previously
SIFS_EPERM	Operation is denied permanently
SIFS_EIO	An I/O error occurred
SIFS_ESERVER	Internal server error

On success "fsync" returns zero, otherwise one of the error numbers listed above is returned.

A.2.19 Lock

Description:

Manage file locks for the given file. These locks can be held on the whole file or just on a part of it specified by a begin and end offset. This request is also only valid on previously opened files.

Request:

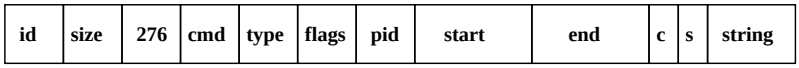

Figure A.43: lock request

type	description	length in bytes
cmd	get and set commands	4
type	Type of the set command	4
flags	flags which are currently set	4
pid	process id holding the lock	4
start	lock begin offset	8
end	lock end offset	8

The "Cmd" field of the lock request can be set to three different values: 0x1 to get lock information about a file, 0x2 to set a new lock and return in the case of an error and 0x3 to set a new lock and wait if another process holds a lock on the file. If the server should wait until an existing lock is released, it will not return a reply until this happens. If a new lock should be set, then a type have to be specified. Possible values are 0x0 for unlocking, 0x1 for a read lock and 0x2 for a write lock. The flags as well as the process id must be set to zero and will be overwritten by the server. Moreover, the lock can be limited to a region of the file by specifying the start and the end offset.

Reply:

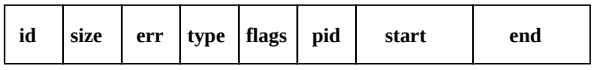

Figure A.44: lock reply

error number	description
ENAMETOOLONG	Pathname "string" was too long
ENOENT	The file was not opened previously
ETEMP	Currently no more locks are allowed
EACCES	Another process already holds a lock
ESERVER	Internal server error

On success "lock" returns zero, otherwise one of the error numbers listed above is returned. The reply also contains the values sent, but they may be modified by the server.

The pid (id of the process holding the lock) returned should be unique on the client and server. This may lead to some problems if processes resides on another computer because there is no global process space, but usually only

two bytes of the pid are used. Thus the remaining two bytes may be used by the server to describe the computer where the locking process resides.

Appendix B

Kernel-sifsd transport protocol description

This sections contain the description of the message types used for communication between the kernel filesystem driver of the client operating system and the client daemon (sifsd). Also the rules are described when a particular message is allowed and who may send it.

B.1 Connect

The connect message signalizes the client daemon to initiate a connection to the remote SSH server. Thus, it includes the credentials to authenticate itself to the client daemon (sifsd). This message can only be sent by the kernel.

Figure B.1: connect message

The payload of this message consists of two strings: the username and the appropriate password used for authentication. If another method instead of a username/password combination is used like the public key mechanism, then the password field contains a random string. It is provided by the user through the mount command as an option. The leading thought behind

the idea is to authenticate the kernel-sifsd (client daemon) connection and providing the shared secret to the client daemon. This prevents an attacker from connecting to the client daemon, establishing a connection to the server and getting access to the files owned by the user.

B.2 Disconnect

The disconnect message causes the client daemon to shut down all connections and exit. By unmounting the filesystem, the user started the clean up procedure where the disconnect message is a part of it. This message can also only be sent by the kernel to the client daemon.

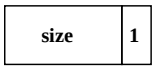

Figure B.2: disconnect message

There are no parameter for the client daemon beside the header.

B.3 Data

Because the client daemon is responsible for forwarding all data transfers between the SSH server and the kernel, a data message is necessary. It can be sent by both parties (filesystem driver and sifsd) to each other.

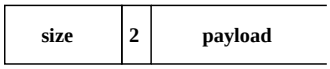

Figure B.3: data message

The payload of this message contains the data, which should be sent to the SSH server or to the kernel. Its maximum length is limited to 4 GByte, but normally its size is limited by the used buffer size of the kernel or the client daemon. The data is transported as provided by the sender. No modification is allowed.