Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Jan Záloha

Stability and security of ZlomekFS

Department of distributed and dependable systems

Supervisor: RNDr. Vlastimil Babka
Study program: Computer Science

2012

I would like to thank my supervisor, RNDr. Vlastimil Babka, for his valuable advice.

Contents

List of figures

Název práce: Bezpečnost a stabilita ve ZlomekFS
Autor: Jan Záloha
Katedra: Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: RNDr. Vlastimil Babka
email vedoucího: babka@d3s.mff.cuni.cz

ZlomekFS je distribuovaný souborový systém, který umožňuje sdílet data mezi jednotlivými uzly. Tyto uzly mohou spolupracovat ve několika módech. Tyto módy se liší v cachování lokálních dat na klientském uzlu a způsobu synchronizace mezi klientským a serverovým uzlem. Tato synchronizace byla doposud implementována pomocí nedůvěryhodného nezabezpečeného síťového spojení. Současná implementace používá FUSE rozhraní pro komunikaci mezi jádrem souborového systému a uživatelem. K synchronizaci dat dochází během jednotlivých událostí vyvolaných operacemi nad souborovým systémem.
Pro funkci v moderním síťovém prostředí je nezbytné, aby komunikující partneři měli jistotu o důvěryhodnosti svého protějšku. Tato práce implementuje důvěryhodné spojení mezi klientskou a serverovou částí souborového systému. Navíc rozlišuje mezi klientem typu stroj a uživatel.
Další důležitou částí práce je korektní průběh synchronizace dat a přesné definování sémantiky sdílení souborů.

Title: Security and stability in ZlomekFS
Author: Jan Záloha
Department: Department of distributed and dependable systems
Supervisor: RNDr. Vlastimil Babka
Supervisor's email: babka@d3s.mff.cuni.cz

ZlomekFS is a distributed filesystem which is able to share data among nodes. These nodes can cooperate in various modes. Modes differ in caching of local data at the client side and in the way how data is synchronized between the client and the server. This synchronization has been implemented by an unsecured and untrusted network connection until now. The current

implementation uses FUSE interface for the communication between the kernel of the filesystem and the user. The synchronization is triggered when the user performs a specific operation above the filesystem.

For well functionality in modern network environment it is necessary to both communicating partners to be sure about the identity of the second end of a communicating channel. This thesis implements trustworthy connection between the server and the client part of filesystem. In addition, it separates machine and user type of client.

Next important part of the thesis is correct data synchronization and precise definition of sharing semantics.

1 Introduction

ZlomekFS is an open source distributed filesystem. It is able to transparently share data among oriented graph structure of nodes [1]. Data are organized in files placed in a directory tree structure and all standard POSIX operations are allowed above them. Main unit of shared data is called volume.

Volumes are organized in a tree structure. Each volume has one node, which is on the top of the volume's hierarchy. This node must have its local cache, the source image of the volume, which is then modified by all nodes in the infrastructure. Connection between server and client has been implemented as insecure TCP channel until now. It means that there is no way how to check peer node's identity. In addition, the connection is susceptible to various types of attacks. These attacks could result in stealing or modifying data by an unauthorized entity.

The filesystem access rights management has been designed in an extraordinary way. The filesystem has a mechanism which remaps users and groups valid at one node to users and group numbers valid at the distributed environment. This information is stored in a metadata structure which is passed through network with the content of the file. Then another node remaps them back to locally valid users and groups. At first sight it works well. But at second sight, when a user mounts volume, all content of the volume can be cached (or accessed) by daemon launched by the local user. It is because server does not check any permission to shared file. So anybody can connect to server with a program implementing ZlomekFS network protocol and access all data, even data which should not be accessible to them.

Next important issue is sharing semantics. When a file at a node is opened, the data contained in the file has some versions. Handling of versions is called semantics. One way, how to solve this problem is stateless semantics. Stateless semantics is implemented for example by NFS. It means, every operation above file is interpreted as atomic file open, the actual operation, file close [2]. But this approach to file operations is not transparent to an application working above the filesystem. For example the file can be removed (unlinked) by another application between two reads performed by the first application.

And there is not any way how to solve this problem without loosing the stateless semantics. Because the access to filesystem is stateless, number of opened filedescriptors can not be held to determine when the data contained in the file should be released.

To ensure transparency of the filesystem to higher layers, there should be a mechanism which correctly handles opened filedescriptors and performs open and close operations independently on other operations. But another problem lies here. After a process opens a file, then another node can push a new version of a file to the node. It could result in strange behaviour when one part of data is read from one version of the file and another parts from another versions. This could not be a problem when it is desired behaviour. But in some cases it represents a serious problem. For example, when the file contains some snapshot of data.

In the old implementation of ZlomekFS this issue is solved in a special way. When a node does not have local cache, the operation is performed on a nearest cache in volume tree structure. So this case is not interesting. But when a node has its own local cache, read and write operations differ. When a read operation is performed, before own read from file, the system tries to synchronize the data at the local cache with the version at master. And when write is performed, no synchronization is done until the file is properly closed. This behaviour is useful when a file is shared from master to client node and the application expects this direction of data flow. But in other cases it could cause many serious problems.

But there exists a way how to solve this problem. Concurrent read and write operations at one node should not be treated in a special way, they occur in all other filesystems. But when a newer version of file is created in the system, it should become accessible to each process opening the file. But each already opened filedescriptor should point to the version of file that was present at system when it was opened, and ignore subsequent updates from other nodes. This semantics was described in [1], but the implementation did not work well.

Next issue is that each file at ZlomekFS is represented by filehandle, data type `zfs_fh`. This filehandle describes how the file should be accessed, which volume is its volume and its other properties. One of the most important property is i-node number, which is treated as volume wide unique identifier of image of the file in local cache.

This way is correct because UNIX-like filesystems (e.g. ext3, ext4, etc...) always gives unique i-node number to each file created on it. But metadata containing i-node number is exported to client. Then the i-node number is used as an identifier of the file, when synchronization of file is performed. In addition, the architecture of ZlomekFS does not permit updating filehandles shared with other nodes.

This property of ZlomekFS does not permit changing i-node numbers during operations. Most important issue of this limitation, as described in [3], is related to the performance of truncating file to zero length while preserving its old version.

1.1 Goals

This thesis should describe and implement security model for ZlomekFS. This security model should describe the behaviour of server side and client side in all possible scenarios, including cached and non cached mode. It should differ operating mode when the client application is launched by a single user or when it is a trusted machine. In case of a trusted machine, the machine should have access to all data stored in ZlomekFS. A single-user instance should have access accordingly to his identity.

Next important issue to solve is session semantics. Current session semantics does not behave coherent when client or server write operation occurs. This will be fixed so when one version of file is already opened at a node, all operations will be done above this version.

At last the old implementation contains implementation errors that affects the stability of all system. So this thesis should fix as many of them as possible.

1.2 Structure of the Thesis

Chapter 2 describes how identity verification and secure communication between nodes could be solved. It also shows how handling client type differs and how user membership in groups could be solved.

Chapter 3 gives an overview of session semantics and shows which advantages and disadvantages possible session semantics possess. It also describes chosen implementation including its properties and possible future enhancement.

The chapter 4 is about issues tied to capabilities and internal filehandles. Because of changes described in chapter 3, there apeared a necessity to fix some behaviour issues of capabilities. In addition, the issue of i-node number directly stored in filehandle is discussed there.

Also another minor changes to the architecture of the daemon have been done. They have been done because of fixing the stability of the daemon. The way how and why they have been done is described in chapter 5.

Because the changes in implementation require more configuration data, in chapter 6 new configuration options are described. Because of chosen security model, it is necessary to generate certificates for clients machines and users. Doing this is also documented here.

The last chapter 7 is aimed at evaluating results of this thesis. It also concludes main decisions and shows possible future enhancements of ZlomekFS.

2 Security

As described above, the old implementation of ZlomekFS does not use any security model. When client connects to server, only a protocol handshake is done and the connection is already usable for data transfers. Standard communication protocol through network is connected TCP protocol over sockets. These sockets can be created over IPv4 [4] or IPv6 [5] protocols.

2.1 Basic security issues analysis

First take a look at protocol IPv4. It is for now the most spread protocol. See its datagram format in Figure 2.1. The whole datagram is not encrypted by default because IPsec enhancement of this protocol is not its native part. Simply can be seen, many attacks can be performed to this protocol. The easiest are network traffic sniffing, stealing identity, data integrity change or man in the middle attack, when the basic networking set-up is done.

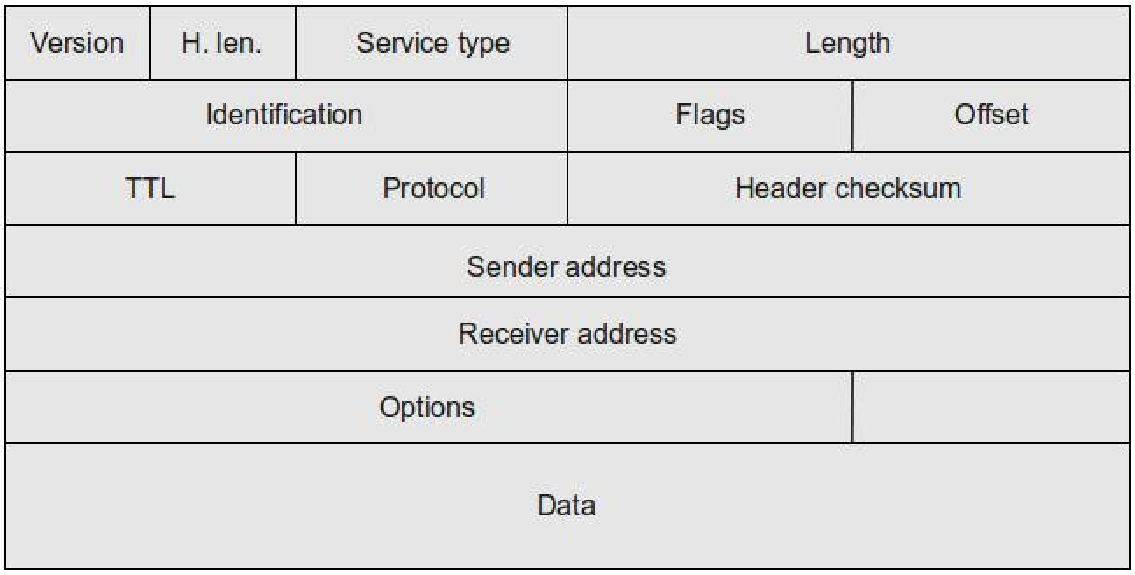

Figure 2.1: IPv4 datagram

Network traffic sniffing can be performed somewhere on the network path between connected nodes. The attacker only listens to network traffic and decodes data transferred through network. This way he can easily read private content of files or directories in case of ZlomekFS.

Stealing identity is very easy, too. The easiest way is running another instance of program implementing ZlomekFS network protocol at compromised node. This way attacker can read full content of any exported volume and in addition he can modify any data stored in such a volume.

When data is passed through network, it goes through it in an unencrypted form, checked only by checksum generated by algorithm whose only input is the passed data. So any network node present on the path between communicating peers can modify data passed in a message and then count again its checksum. When a message modified in this way arrives to its destination, there is no chance how to recognize its modification.

Attack man-in-the-middle is a special case of these attacks. Some node on the path between nodes behaves same way as the server for client side and same as the client for server side. So he has full control above data passed through network and can transparently modify or read them.

These attacks can be solved by using IPv6 which should implement IPsec enhancement by default but, as written above, it is not as widespread as IPv4. Another solution is very careful configuring of network and implementing IPsec above IPv4. But it is a quite difficult activity.

And here is another issue: these solutions can lead to trusted machine and connection to it. But we can not do any conclusion about the program launched on it. It could be a malignant program launched by an attacker. Even it can be correct implementation of ZlomekFS but not launched by an administrator. It could be launched by a common user. In the old implementation of ZlomekFS this instance of filesystem daemon have full access to all data exported by available volumes.

This could be a large problem. It is because in cached mode, all data available at one volume is transferred to this node and it could lead to massive network traffic. Another problem rises when a user uses somehow changed or hacked program or launches proper implementation of ZlomekFS in debugger. In this case, he can access all data even the data that should be hidden to him.

Because of this, there should be a mechanism which recognizes type and identity of client connected to server. The question is how to divide and recognize types of clients. One type is user who connects to server and acquires data. Data accessible to him should be only the filesystem entries belonging to the user or belonging to a group where the user has a membership or public data accessible to all users. No other data should be transferred to such a client. On the other hand, there should be nodes which contain all data shared by volume.

So basic goals in the area of security are set: The communication between nodes should be resistant to traffic sniffing, data modification and man-in-the-middle attack. The communication protocol should be resistant to basic cryptanalysis. And the identity of communicating peers should be verifiable and, accordingly to it, only a limited set of data should be accessible to each node.

2.2 Other security issues

Except the problems described above, ZlomekFS can be vulnerable in other ways. The biggest problem is local cache stored in filesystem.

2.2.1 Local cache

There are many other possible attacks to ZlomekFS. Very vulnerable is local cache containing data. When wrong access rights are set to the cache, unauthorized user can access these data. So it is necessary to discuss managing access rights to local cache.

Next dangerous problem could be the presence of local cache in unencrypted shape on disk. Because when the disk is mounted under another system, the cache can be accessible to attacker. But this issue can be simply solved by the user. The user can place the local cache to a virtual encrypted volume, for example by Truecrypt [6].

2.2.2 Data in memory

Naturally, when the ZlomekFS is launched, its working set [] data is in its virtual memory in unencrypted form. This can cause two problems. When some data is swapped out, they are stored in a persistent form. An attacker can read them from the disk when he starts another instance of operating system and then dumps the content of swap partition.

This vulnerability can be avoided by disabling swapping or some other way by encrypting swap partition for example by [7] or [8]. In this case there is another problem: how a and where store key for encrypting swap. It is a classical chicken-egg problem, But it can be also solved by, e.g., using a key stored at some removable device or by using the biometry of user (or administrator).

Another problem could be raw data in memory. Today attacks based on stealing data directly from main memory chips are known [9]. The design of this attack is in stealing memory modules from a running machine and their very fast cooling (e.g. by liquid nitrogen). When the modules are deeply frozen, they can hold data up to several minutes. But this issue is outside the scope of this thesis.

2.2.3 Not solved attacks

Another attacks can be performed to the shape of network traffic. When an attacker has permission to the network media connected to node, he can read raw data sent through the media. When the data are encrypted in the correct way with a good encryption algorithm, it is supposed to be very difficult to restore the image of the ciphertext or the encrypting key [10]. But when the encryption algorithm is deterministic and known, the attacker can often easily restore for example the length of sent data. From the length of the data some conclusions on the type of data can be done. In some cases this could be undesirable property. This issue could be solved by inserting (pseudo)random chunks of data to output stream. But this solution could lead to high network traffic, so this question is not solved here.

Another problem can rise from the external behaviour of system. For example the heat emitted by its components or the consumption of electricity can be a source of information about system activity. But these issues require much deeper analysis than this master thesis and do not represent a serious danger in planed filesystem use.

Next some conclusions about system activity shape can be done by observing activity indicators such as hard disk or network activity LEDs. These indicators could be affected by generating (pseudo)random traffic or by updating drivers of such devices. But generating additional traffic can lead to high network traffic as described above. It stresses components and could decrease system response time.

Another serious issue is DoS (Deny of service) attack [11]. This can be performed by many ways. First aim is networking stack of machine. Solution of this is a part of the design of operating system, so it would not be solved in this thesis. An attack to network infrastructure is similar. Again it is out of range of this thesis. Next serious issue is sending not valid data to machine by proper client, e.g. when it crashes. This can lead to serious problems and crash or malfunction of node. Some steps to avoid this are done but a very deep analysis and probably rewriting of a large amount of recent code and adding a large amount of new one should be performed to be sure, this issue is not already solved.

Last main area that could lead to DoS is wasting node resources. Some operations performed on node require system resources as memory or CPU time. It can lead to malfunction of the node or to severe slowdown of the node. The biggest problem is to answer how much resources one node could acquire. In distributed environment of ZlomekFS it is hard to say how much resources one node could acquire. Because when the net of ZlomekFS is formed by two nodes, the amount of resources available to one node differs to case when the root of a volume is processing peak traffic of requests from a very large network.

Next it is important to realize how to manage resources in environment where no assumptions about nodes availability can be done. Nodes are connected and disconnected more less randomly. Even a node which is available for communication can disconnect because of lack of its own network resources. Some proposals how to manage this question are done in chapter 4.2 but this problem is left open.

2.3 Possible trustworthy solutions

There are many ways how to check identity of the peer and/or encrypt the communication channel. The easiest way is assigning a key to each node and publish the keys systemwide and use them for the digital signing of passed messages.

2.3.1 Signing messages

By using this mechanism message trustworthy can be reached. Each node gets a key which is used for message signing. Before sending through network from each message and the key is counted a hash. The hash is appended to the useful data and this message is sent through the network. When the message arrives at the second endpoint of connection, the data is picked up from the message. Corresponding key is selected accordingly to the assumed peer. And the hash is counted from key and that data. The result is compared to the hash appended in message. When they are equal, it means that the message has not been changed during network transfer and the origin of data is verified. This solution introduces three obvious serious problems. First, the data are sent through the network unencrypted, so an attacker can read its content. Second problem is distributing keys through the system. Because of preceding issue, keys must not be sent in unencrypted form through network. Suppose that the keys are sent some way through an encrypted channel but before owning them, it is not possible to check whether the message providing key is from a trusted source. So there must be another trusted communication channel for distributing keys. In addition, each node should know keys for each other node which could connect to it. But let us suppose that this problem is solved.

Then another question rises: should symmetrical or asymmetrical hash function be used? When symmetrical hash function is used, both sides use the same key for signing and verifying messages [10]. In trusted environment this would not be such a serious problem. But in the environment of ZlomekFS where common users would have access to keys used by another nodes, it would lead to serious problem. Users could steal identity of other nodes.

So for this reason an asymmetrical hashing should be used. But here is one serious issue described above: messages are sent unencrypted and attacker could read their content. So there should be a way how to hide data in messages.

2.3.2 Encrypting messages

This data hiding can be performed by message encrypting. This should work that way, one node before sending message encrypts it, sends it through network and the second side decrypts it. Because encrypting is view form set of plaintext to ciphertext and decrypting is view from ciphertext to plaintext and these sets have same cardinality for common algorithms (Blowfish, DES, 3DES [10]), another attack to this solution can be performed.

When data is sent through network, an attacker can modify the message and this way the image for decrypting is changed. When the decryption procedure is performed, it does not recognize the message change because of the same cardinality [10].

For this reason, a technique for recognizing such attacks must be implemented. As described above, message hashing is suitable for solving this problem. So after encrypting, a signature will be attached to each message and it allows the second side to recognize unauthorized modifying of the message. Even the same key can be used for encrypting and signing.

But this solution also brings problems with key management. These problems are the same. When a conventional fast block cipher is used, decrypting and hash verifying keys must be distributed through system.

2.3.3 Key distribution

As mentioned above, the key management is a serious problem in distributed environment. In case when each possible connection has two pairs of encrypting (and signing) and decrypting (and verifying) keys and the structure is a tree, it means distributing and sharing many keys. Undesired problem then occurs when a new entity is added to system. It means distributing its verification key to all its neighbours and backwards installing verify keys of its neighbours on it. When this minimalistic approach is done, it makes it difficult to dynamically change topology of the network.

Implementing a smarter key management would be a better solution. For example, public key infrastructure [12]. This key management is designed for using asymmetric cryptography. Each encrypting entity is represented by public certificate and private key. These two keys are closely tight together:

Transformation of message by one of them is reversible by transformation by the second one. In the simplest case the public certificate is used to encrypt data sent to owner of private key. Let us suppose that this operation is irreversible, so the public certificate can be shared among all communicating partners. It can be done for example during establishing connection. So when the message is passed to node, it is transformed by its public key (certificate), transferred to destination node and there again transformed by private key. The result of this transformation should be the original message.

Another important property of this key management is its usability for digital signing [12]. When a message is passed from one node to another, there is counted a checksum by a known deterministic algorithm. Then the checksum is transformed by a source's private key and appended to message. When the message arrives to destination, the transformed checksum is again transformed by public key. From message data is same way counted hash and compared to result of transformation. If the message has not been changed, these two data chunks should be the same.

This is very useful also in key distribution. Suppose that exists one trusted private key and its public certificate (call them A and A'). Another pair of a private key and a public certificate (B and B') can be generated and verified their correctness. Then some special data can be added to B' and a hash of this data object is counted. The hash is transformed by A and the result is added to updated B'. Then the pair $\langle B, B' \rangle$ is assigned to some network entity C and public key A' is published. When any node D connects to C, entity C first sends to D certificate B' and D can verify its correctness by performing procedure described above.

This key management also allows creating trees of trusted entities like in the Figure 2.2. The root of this tree is called certification authority. Its special property is that its public certificate is signed not by a superior key but by its complementary private key. This key pair is then used to sign and verify first level of other certificates. These certificates can be user certs. It means, they are not allowed to sign other key material. But another can contain special privilege and could be used to sign new certificates. Then after signing by such certificate, its verification certificate is appended to just signed entity. This way a tree structure with a very useful feature arises: only the root certificate authority's certificate is needed to verify any signed certificate. It is because first is verified first level sign. When it is considered trusted, it can be used to verify next level and so on.

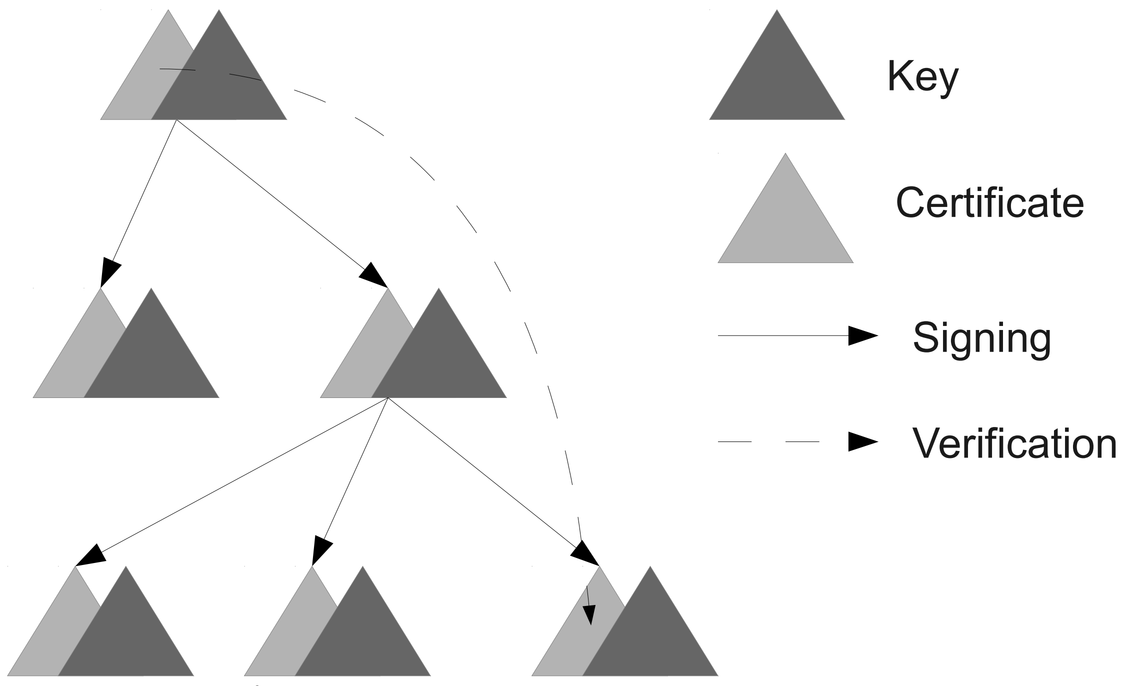

Figure 2.2: PKI infrastructure

2.3.4 OpenSSL

All these requirements are satisfied by X.509 standard [12]. This standard allows hierarchical public key infrastructure and, in addition, revoking certificates. It means that when a key is compromised, it can be invalidated. The invalidation is performed by a list of revocated certificates (let call the list CRL – certificate revocation list). This list is signed by certification authority. When a certificate is considered untrusted, its identifier is added to this list. Then the list is signed by the certification authority and distributed to all nodes [12]. The distribution channel may not be secure because of signature of the list which provides trustfulness.

When there is established communication channel between two endpoints, they show their certificates to the peer. The peer looks into the CRL and when the certificate is there present, it rejects further communication.

This behaviour is very useful in environment where it is supposed that the private key of entity can be stolen or other way compromised. And the environment of ZlomekFS where each user might have its own certificate meets these properties.

So there should be an authority: the administrator of whole system who will issue new certificates and also issues and updates the CRL.

Most common framework implementing the X.509 standard is probably OpenSSL [13]. This protocol can be set above TCP/IP protocol, so there are necessary only minor changes in the old implementation. But OpenSSL is not only a layer providing authentication of communicating nodes. It also encrypts the connection between nodes by cryptographic communication protocols. Available protocols are SSL v2, v3 and TLS v1 [13]. These protocols should provide enough amount of security for purposes of ZlomekFS. When using sufficiently long asymmetric keys (for establishing of connection) – it means about 2048 bits, the probability of compromising the communication channel is very low for now [10]. In future, the length of this key will rise but it can be very simply replaced by another key.

Quite pleasant property of OpenSSL is the fact that it is not only a library for securing network connection. It is a complex package including many utilities for managing key material and generating support files. Because of this OpenSSL is a good choice for securing ZlomekFS.

When X.509 format of certificates is used, they can contain another information about its holder, issuer and validity. Very important entry is their period of validity. It is very unfortunate solution to have a certificate with unlimited validity period. It is because computers are still rapidly faster and keys providing enough security today will be totally obsolete after a few years (maybe earlier).

Next important entry is also identity of holder. From this field, it can be get by trustful way, who is the peer. X.509 format also offers some optional fields, which can be used to determine the type of peer node.

2.4 Node types

As described in chapter 2.1, nodes created by single users can access only the data that the user is authorized to access. But when all nodes would be only user nodes, it would rise a problem: Each part of filesystem would have some data set but nowhere would be all image of a volume. So when another node (user) would connect, it would prove its identity but the data owned by the user would be stored at node with the same user identity. It would mean: all users should have their private data set in their own local cache and share only public data.

This approach does not meet the idea about distributed filesystem which has a master node where a user connects, downloads its data set, works on it and then stores the data back to the repository. Some time later again he checks out all data set from another node and can continue his work.

So there should be some entities which contain all data. But as discussed above, it should be not the user. So it is necessary to involve another type of node trusted to contain all data.

2.4.1 Machine type node

As any other process, the instance of ZlomekFS daemon on a machine type node has to run under a UNIX user account, and the question is which one. One of the solution is introducing special users whose task is just storing data into the cache. It would prohibit access to data in the cache by other users. But here is another problem. The mechanism of storing file metadata is solved in a quite specific way. When the file is created, its owner `uid` and `gid` [1] are translated to systemwide known numbers, stored to metadata of this file and the file is stored to local cache under its permissions belonging to its creator.

This method has one serious disadvantage. In case that instance of ZlomekFS manages files belonging to another user, the mechanism of saving file access rights does not work. It is because of changing access rights to file the process must have appropriate privileges [14].

Also here is another way how to treat this issue. It can again be done by defining a special user and group where this user belongs to. But in this case, all local cache would belong to that user and all operation above access rights would be redirected from filesystem to metadata. But this solution requires a very deep change of the old implementation of ZlomekFS. More or less it would mean the change of all FUSE interface.

Another solution is launching the daemon of ZlomekFS as a superuser or another user which is allowed to change privileges of files. In this case it would be good to declare the whole machine and any operation done by it does not have to be checked when propagated to another machine.

It is because the type of installation. In the case the user connects the client to ZlomekFS infrastructure, he has his own certificate and possibly his own implementation of ZlomekFS daemon. But when the daemon is launched under a special user which is created (or another way trusted) by machine administrator, the administrator can do steps to avoid stealing its identity. For example storing keys in a secure location.

This approach also has another important advantage. When FUSE filesystem is launched under superuser, the option `allow_other` can be passed to it. In this case the content of ZlomekFS will be accessible to all user under proper access rights. It is possible to pass argument `allow_other` even in common-user mode, but it requires changes in the `/etc/fuse.conf` [15] and they are systemwide valid. So this way would affect, probably undesired, behaviour of the whole system.

Here a question arises how to differ the described type of node from the node created by a root at a machine for his own purposes. In addition there would be another undesired property: the root could be mapped to a common user at ZlomekFS infrastructure, so each connection would require a mechanism how to ensure that peer is really machine's superuser.

Much better solution is using certificate data fields to store information about the type of node which owns it. So now at least two types of certificates are needed. One for fully trusted machines and one for individual users. Here is a question whether another type of certificate would be needed.

It is sensible to expect that these two cases are extreme. A smaller unit which can get access rights could be defined as an intersection of directory tree and user's access rights. But this way have not any benefits. Similarly larger units than one machine need not to be considered, too. So the last question is whether there is an entity which might be important and is not user nor machine. But ZlomekFS and FUSE are designed for two basic purposes: to mount filesystem for one user and nobody other can reach it or to mount filesystem for all users at a system level. So no other entity should not be introduced into the infrastructure of ZlomekFS.

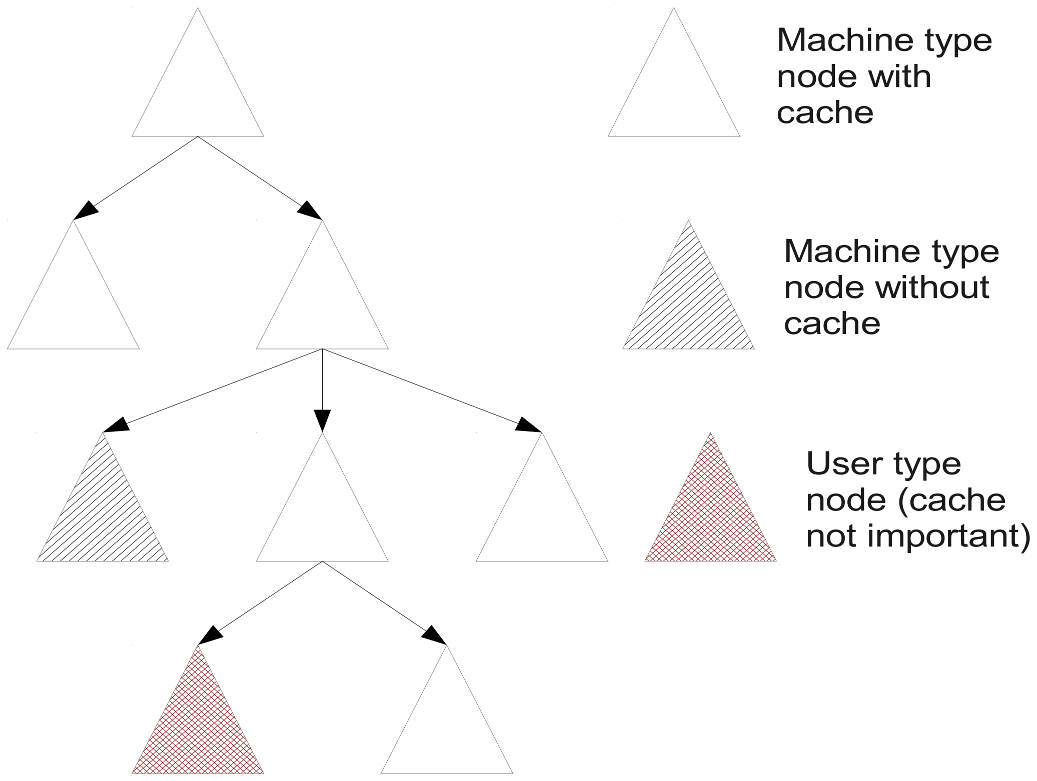
Machine type node with cache
Machine type node without cache
User type node (cache not important)

Figure 2.3: Network schema

This solution brings one important property of ZlomekFS: the shape of infrastructure tree is more precise showed in Figure 2.3. The root and non-leave nodes should be trusted machines with all data accessible. Leaves of the tree are nodes which can be common users or fully trusted machines. And there should be a mechanism which prohibits connection of node to an instance with user privileges because only incomplete set of data would be accessible. O n the other hand, there can be scenario where a user mounts his data at a “router” which is connected to a ZlomekFS infrastructure. Then he connects another local node to such a “router”. Then from the architecture of ZlomekFS, there would not be a problem. But because because this this approach would lead to major architecture changes, this behaviour is forbidden.

2.4.2 User mode connection

As described above, these nodes are placed as leaves in hierarchical structure of ZlomekFS. They are intended to make accessible private data to each user. They are intended to work in cached or non-cached modes.

When such a node connects to server, it proves its identity by a certificate. Then every operation required by such a client is verified by the server. The verification includes file permission check. When access rights to file allows desired operation, it is done, in another case it is rejected.

Here rises another problem: what about files that have executable bit set but read bit not set for that user. The user is allowed to execute such files but for executing the file should be transferred to a client node. But when it is transferred this way, the data of the file can be stored also at that machine. It means that the content of such a file can be compromised by that user. But here should be a way how to launch them. This requirement leads to a special limitation. When a user wants to launch an executable file form, a node with only user certificate, he also should have the permission to read that file. This solution differs from the standard behaviour of POSIX system but in order to prevent compromising data it is necessary.

Another problem rises from the local cache access rights management. Files are also stored in underlying filesystem and their access rights are stored in it. From discussion above flows, it is not a nice solution to transform file access rights directly from metadata to local entities.

But when a common user launches the daemon, it inherits his identity, so changing owner and group of file in the cache is not possible. The only possibility is granting CAP_CHOWN [14] to that user. But by granting this capability, effective privileges of that user dangerously rise. So in this case, when using user-type of certificate, the executable of ZlomekFS should have set the setuid bit and should be owned by a user who is capable to change file owner.

At first sight, it should be dangerous, to launch such a large project with superuser privileges. But this problem can be simply solved with a chroot utility. The ZlomekFS own daemon will be stored without the setuid bit enabled. But another utility, call it zfs_launcher has this bit enabled. So when it is launched, it again creates command line accordingly to arguments passed to it and launches ZlomekFS with chroot somewhere in user's home directory.

In case the implementation of ZlomekFS does not contain a fatal bug, this should prevent possible attacks to local system. A fatal bug is a bug which allows replacing the code of running program by a harmful code. This attack, called stack overflow, is performed by sending data of wrong length to a program input. But OpenSSL is used for network communication and all buffers are well bounded. File inputs are again bounded. It is important to prevent stack overflow. Because when it happens, the chroot jail can be broken [16]. So if these conditions are met, there should not be done an attack dangerous to all system to ZlomekFS daemon when it is placed out of root directory.

2.4.2.1 Group membership

The user is always a member of a group, sometimes not only one group. So here rises a problem how to manage mapping group membership of a user. It is because at one machine, the user can be a member of a set of groups, but at another machine, he can be a member of a different set of groups. In addition, there is a possibility that, at a node, there does not exist mapping from systemwide group to local group.

This issue can be solved in many ways. One of them is restricting the user membership at all nodes connected to ZlomekFS. This approach is not scalable. Any group membership change at one node should be propagated to other nodes before the user connects there. In addition, it is not transparent to nodes' administration because there should be some restrictions to group membership at some nodes. Simply said, the environment may not be homogeneous.

So another solution should be chosen. It could be based on information about group membership present in data used by the filesystem and automatically spread. First such data can be certificates. The information about group membership can be stored in them. In real it means, the information of group membership can be stored in an entry contained in certificate in the same way as the user identity.

But when all systemwide groups would be stored in a certificate, it would bring many problems. When the group membership changes, a new certificate must be issued. In addition, the old certificate must be revoked. So this way is not the best. A better solution would be storing group membership information in a separate file stored and distributed by filesystem. These files should be accessible by all nodes but changeable only by administrators. For this purpose *configuration volume* [1] could be used because this volume has proper access rights: Anybody can read it, but only the user mapped to ZlomekFS administrator can modify it. In that volume a file containing information about group membership could be placed. And the certificate may contain only an index to that file, which allows group membership searching. This solution also allows to have one entry per certificate and the entry is not dependent on machine where it is used. At this machine the mapping of groups is then defined by its entry in the mapping mechanism.

The *configuration volume* also has another important property. When any change is performed on it, the ZlomekFS mechanism notifies all clients about such a change. When a change is reintegrated to server node, closing of that file triggers the same mechanism at its clients and further reintegration to higher levels of hierarchy is performed. This way important setup information is spread very fast through all infrastructure.

Next reason why it is a good way to treat this information is the fact that all information about ZlomekFS users, groups and its mapping is also handled this way. So this solution is logically consistent with the preceding implementations.

2.6 Certificate revocation

Because of usage of this system by many user, accidental identity compromising can be caused. This can happen by losing user's private key. In case of a machine type certificate it is an extremely serious problem because the new owner of key gains full control of all data shared through the system. So here must be a mechanism of secure storing files containing key material.

This problem is solved directly by placing machine certificate to configuration directory with special access rights. These rights should be adjusted so that any other user can not reach that file. But here is another issue: storing of user level certificates. The user should be allowed to transfer his certificate among all machines that he can use.

This type of behaviour brings a higher risk of losing the identity. So there should be present a simple way how to invalidate X.509 certificate as described above. The OpenSSL framework brings full support of these actions but another problem is how to distribute this information.

The simplest way is providing CRL with other identity data on startup of node. But this solution brings a problem how to obtain the CRL list. It would mean, before each startup the user should use a service (e.g. web or another maybe proprietary service) to obtain this list. Again, this connection should be trusted because in another case an attacker should provide obsolete CRL list. Of course CRLs also have limited time of validity, so this should not be such a problem. It is because obsolete CRL would be valid only for a short time. But in distributed environment of ZlomekFS when nodes can work disconnected for a long time, it would bring a problem. In some disconnected areas all CRLs would become obsolete and no communication could be performed.

Better way may be to generate CRL with a longer period of validity and distribute it with data of the filesystem. But again this solution has disadvantages. The main reason is when a machine type of certificate is compromised, as shown in Figure 2.4, all its subtree has a serious problem.

Consider a situation when the system administrator realizes that a machine type of node has been attacked and its certificate is stolen. So it takes root certification authority's key and compromised certificate and updates the CRL. Then the CRL is copied to proper path at the ZlomekFS directory structure. There could be three basic types of node where this operation should be done as shown in Figure 2.4.

Suppose when a certificate is stolen, any assumptions about identity or correctness about that node could not be done. Only in case when the CRL is updated at compromise node, the identity of the node is known, but still there could be a problem about the correctness of local ZlomekFS implementation, so it could be unable to share the CRL.

When the new CRL is installed in the subtree where compromised node is root, it should by installed though that tree but it is only the first sight. The behaviour of such a damaged node is undefined, so it can update version of file holding CRL and this way cause a conflict.

A very similar situation is when the new CRL is installed from outside of such a tree. Again the malicious node can cause conflict and there can't be done any assumptions about sharing CRL through network. So this way seems not to be the best solution of the problem.

But there is not any other way how to obtain CRL. Only two types of communication channels are available: by own ZlomekFS infrastructure or by another channel. Both of them have similar problems. For the first case a restriction must be defined. Especially about behaving and administrating of nodes with machine type certificate.

The discussion above brings an important result: Distributing CRL through ZlomekFS is not a good way how to manage revocated machine certificates. But on the other hand, when an attacker reaches the private key of machine type node, it means that he has had access to secured store of the key material. But this may happen only when administrator's credentials are broken or the daemon is hacked. But in these cases the validity of the local cache or data sent to other nodes can be corrupted and the topology of the ZlomekFS net can be affected. So stealing private key of machine key is only a subsequent case of a much serious problem. And the solution of this is not only replacing the key with a new one and then to revocate the old one. First, all such attacked nodes should be repaired, correctness of installed version of ZlomekFS, other programs and settings should be checked. Then a new certificate for this node should be issued and the old one revocated. Finally, after systemwide synchronization of CRL, it should be checked whether no conflicts occurred and the synchronized version is equal to just declared version. Good news is that because of a high level of securing (described in 2.7) machine mode nodes, this case should not happen.

But another case is user type node. The user takes his key material with them, so stealing it is not so difficult. But revocation of such a certificate is easy. Important points are: the client mode machine is a leaf in the ZlomekFS topology tree and it is typically not allowed to modify configuration data. In case that it tries to do that, reintegration of such a file will be rejected by its first neighbour and systemwide settings will not be corrupted.

So when compromising of user type certificate occurs, it is enough to generate new CRL and update the old one. The change will be spread through the infrastructure and communication channel to the compromised node disconnected.

2.7 Implementation details

The proposed improvements to ZlomekFS naturally requiree significant changes of implementation. The most important is the change in network stack of ZlomekFS.

2.7.1 Network stack

In previous implementation the networking stack was solved as a simple filedescriptor placed in structure `fd_data_t`. This structure was then passed to another calls and native POSIX calls were performed above the integer filedescriptor. Also as a side effect the integer nature of the filedescriptor was used as an index to field of structures `fd_data_t`. This is possible because the filedescriptor is a small non-negative number.

But when OpenSSL is used for securing network communication, it must be possible to manage network communication by it. There are two main ways how to solve this problem. One is rewriting all network module accordingly to OpenSSL requirements. The second is based on observation that OpenSSL is only a wrapper above sockets and provides similar functionality and even the calls are very similar. So the network connection has been redesigned as a virtual object which provides all necessary functionality to client application. These functionalities are reading data, writing data to connection and support functions as binding, connecting and others.

Because network connection is not now just an integer, but a dynamically allocated object, it must be properly deallocated when it is not further needed. It would be a simple operation in environment where only one thread has access to that structure. But ZlomekFS networking module is designed to be multithreaded and this structure is shared between threads. So the technique of reference counting to object has been chosen. It is fast and reliable. When a thread gets pointer to the object, the reference counter is atomically increased, when the thread releases the object, the reference is decremented. When the counter reaches zero, it means no thread knows the pointer to it, so it is further unreachable and it can be deallocated.

Another problem in the networking stack has been statically allocated array field of `fd_data_t`, which describes internally each connection. It has been indexed by filedescriptors assigned by kernel. A nice solution would be to dynamically change its size. For this purpose the special indexing layer bound with network connection has been designed. But during a deep analysis of the module important findings were made. Not only filedescriptor is passed to other thread but just pointer to all structure `fd_data_t`. So here could be done two decisions. The first one is passing a copy of the structure. But the structure is designed for data sharing between threads and copying of it would lead for example to problems with synchronization. It is because it contains a condition variable to manage threads using this structure. And pthread entities should not be copied [17].

The second way is to hold only pointer in the field to the `fd_data_t` structure. But it introduces the necessity of using another reference counting and another synchronization in network module. The reference counting can be solved simply but after analysis of locking scheme of all ZlomekFS daemon, it was considered to leave the old implementation of managing connections untouched. It is better to waste a few kilobytes of memory then to risk a deadlock.

The next necessary problem described above is the secure storing of key material. Especially the machine type key is very vulnerable. So it should be stored in a secure place. There were considered many ways of placing this data. But because of decreasing dependence on other services, storing in local filesystem has been chosen.

Because of this, access rights to configuration files should be precisely checked. When the ZlomekFS daemon is launched, it checks access rights to files containing key material. These files should be owned by the user who launches the daemon and access rights should permit at least owner read access to it. All other rights should prohibit access by group or other users to them. In other cases the daemon will exit.

2.7.2 User mode launching

In the old implementation the way of treating local cache, even then in user mode, should be the ZlomekFS launched with superuser privilege. But it would mean possible security violation. Therefore some steps to avoid such possibilities must be done.

The most important thing is to avoid possible operating system data and configuration change when the process with superuser privileges is running. So the possibility of launching the daemon in chroot jail [14] seems very interesting. It means the virtual filesystem image is created at a filesystem point. At such point any program is then launched and its root of filesystem is just there and it can not affect other parts of filesystem.

Other possible way is to drop the privileges of running daemon. It could be done by changing the owner of the daemon from `root` to another user. The best choice would be special dedicated user `zfs_user` which will be created during system installation. And then the daemon will be launched with privileges of this user.

This approach brings new problems to solve. When such a process would be launched, it can not change directly permissions to a file. It is a problem with the old implementation of cache when access rights are directly stored in underlying filesystem. This issue can be solved in two ways. The first one is granting `CAP_CHOWN` privilege to the daemon. But this would be a serious security issue because when a process has this privilege, it could behave similarly to a superuser process when it changes owner to himself and then change the file.

The second way is to discontinue the way of storing attributes at filesystem level and store them somewhere else. The best way is probably to store them at metadata information and leave files at cache under full control of the daemon. Because of the possibility of creating special files, the process would have granted privilege `CAP_MKNOD`.

This way would solve the main security issue. On the other hand, it would be nice to prevent the daemon to access another data when it gets corrupted. This can be done by moving it to a jail [14].

But this feature of UNIX systems (especially Linux) has some disadvantages. The most important is the fact that all files, including libraries, used by such a program must be copied to the place where the root directory of the jail will be.

First of all, access rights to the root of jail should be changed to `0500`. It is because its owner should have read and write access to it. Then required libraries and configuration files should be copied to the jail. So for doing this, all directories except the `/etc` and `/var` are removed from the jail and replaced by new. It is because ensuring correct behaviour of the launched daemon. The `/var` directory is left untouched because it is supposed that the cache will be placed there. And similarly the `/etc` contains important setup of the daemon.

Other issue closely tied to the previous one is the necessity of presence of all character devices used by such a program. So first, there has to be prepared the environment of the jailed process. But in this environment even some special devices from directory `/dev` must be present. But it can be a problem. This directory is mounted as another device, so making hard links to desired devices is not possible. When symbolic link is made to `/dev/fuse`, it is only a path to a file out of chroot, so it is unusable.

Another way how to provide special devices to the jail has to be performed. The problem can be solved by `mount` utility. It can mirror any directory to a mountpoint when the option `bind` is specified. But just mounting all `/dev` directory to the virtual filesystem would not be a good choice because the daemon will be able to access all devices. So there should be created a subdirectory `zfs_fuse_dev` where hardlinks to needed devices will be present and this directory will be mount to the `/dev` directory in the jail. The command used for this is: `#mount -o bind /dev/zfs_fuse_dev/ ./dev/` . The FUSE daemon can be launched at the jail now.

It is important to say that the utility preparing the jail must have superuser access rights again. This can be ensured by a `SUID` bit attached to it. It would not be a problem because the code of this utility is very simple.

But another issue should be solved. How to behave when the user wants to place local cache somewhere else than the `/var` directory. This problem is very simply avoided by forbidding placing the cache somewhere else in the user mode. It is possible, that all daemon is launched in a jail, which is prepared by a privileged process.

2.7.3 User and machine mode

New scheme of behaviour when the client is trusted machine or not fully trusted user has been introduced. This has brought also a new code inserted to the network module.

Types of certificate are recognized by field nsComment. A machine has its value set to string ”`Machine certificate`” and a user to ”`User certificate`”. Accordingly to the value of this field the access rights check is then launched when an action is performed remotely.

But for doing so, there must be implemented a mechanism which allows mapping users to groups independently to local machine settings. It is done quite simply by the configuration file and data held in certificate.

When the type of certificate is ”User certificate”, it also holds the systemwide number of the user in field `subjAltName`. But this is too small to hold all information about user's identity. From the user type certificate the group membership information about that user should be also available.

Here are two basic approaches to this issue. One is holding all information in the certificate. But it means to issue a new certificate for each group membership change and to revoke the old one. This means the necessity of a sometimes massive communication between the system administrator and the user and to manipulate with vulnerable key material.

Because of this, a better solution would be a table indexed by user systemwide ID and containing the membership information. This configuration file is designed as line oriented. Each line contains user identification followed by the list of systemwide identifiers of groups. This file is stored at *configuration* *volume*, so it can be changed whenever it is needed and is also available to all nodes.

2.8 SSL utilities

Because the filesystem uses SSL framework for communication, it is very easy to use this package of utilities to manage key material. To be user friendly, shell scripts for certificate management are provided with the filesystem.

3 File sharing semantics

Each filesystem should have well-defined rules regarding propagating changes to its files. These rules can differ in many ways. As described below, there are several issues in a distributed environment, where no serializing authority is present [18]. One of the most common network filesystems is Network File System.

3.1 NFS [2] overview

This filesystem has been designed at 1980's by Sun Microsystems. There have been published three newer versions which differ from the old implementation. But the main importance of NFS is in its old versions which use stateless semantics.

It means the system itself does not have any mechanism dependant on client connection. It is because it is divided into two main parts: the mount daemon and the actual NFS implementation. When the client mounts a NFS volume, it first asks the mount server for a handle of entry point to the NFS volume. Of course, it should provide its credentials to the mount server.

When the handle to the entry point is provided, the client can use it to obtain handles to its child directory entries and so walk through the directory structure. But this handle is not only resource for walking the directory structure, it is also used for identifying files. It has the same functionality as filedescriptor in local filesystem. Reads and writes to NFS are identified only by this handle [2]. It means no explicit open or close to NFS filesystem is required. So when the client application does such an operation, it is ignored by server and always succeeds. Any subsequent reads and writes are done above actually closed file.

Requests to these operations are delivered to NFS server and here the file operations are atomically done. Each operation is interpreted as open, the actual operation and close. When client realizes that the operation failed because of a network error or a server malfunction, it can retry the operation until it succeeds. These operations work fine and the semantics is well defined. But the protocol has also disadvantages.

Because it is stateless, the server can not count references to the opened files. It means such sequences of operations like open file, unlink the file from directory structure and using it as a hidden storage do not work.

Quite similar is the case when a file is opened by a privileged user, then process drops its privileges and can work with that file is quite similar. In addition, the NFS can have only one cache (in this case image would be better name), which is in the root of its topology and other NFS servers then only redirects.

But in NFS such problems can not be conflicts. It is because all operations are done above one systemwide image of filesystem. So when an operation succeeds, it should be persistently stored in the filesystem. And when another operation is done concurrently, there are only two possibilities: they are not mutually exclusive and both succeed or one of them fails.

3.2 UNIX semantics

This semantics is designed for local filesystems where the kernel can synchronize each operation. This filesystem has states. Each file before accessing must be opened. Process which has opened the file gets filedescriptor which internally represents the file.

At this moment, the process has access to that file independently on any access rights changes. Even the file can be unlinked from directory structure but process which have opened filedescriptor to such a file can work with it independently on directory structure changes.

All operations above such a filesystem are synchronized by the kernel of underlying operating system. It is because at one machine synchronization entities are effectively implemented. Because of this concurrent accesses can not occur. Suppose we have multiprocessor system where two processes try to access (for example write) a file at one time. The underlying operating system cooperatively with hardware ensures, that one access is done before the other. So modify-modify conflict [1] can not occur here.

The same case is create-create conflict [1] because it is same as modify-modify conflict, only above a file with directory attribute. It is the same for attribute-attribute conflict [1]. Modify-delete conflicts []1 are again not possible here because of counting references to open files as described above.

This semantics is perfect for handling files but it requires very strong resources: synchronization entities or another scheme to avoid conflicts. And this entities are hard to implement in distributed environment [18]. Mentioned algorithms work well in a relative stable environment.

3.4 Immutable files [18]

Some filesystems use a special way for accessing files: immutable files. It means that no write operation is permitted to a file. File is always created with all data and it can be further modified.

When the file should be modified, all its data must be stored somewhere, the file is then removed from directory structure and new file is created instead.

3.5 Session semantics

This semantics again uses open and close operations to get capabilities to a file. In its pure shape it says that when one process opens a file and modifies it, the modifications are not visible to other processes which have this file opened. This allows one important feature: processes if they have their own copy of file, do not need to communicate. They only play with their data and in the end, they close them. After closing all, following opens should see the modified version of a file.

This feature means that changes should be propagated to other nodes after close and only then all following opens should see this version. But all current opens should see their older version of file.

3.6 Comparison of these semantics in the context of ZlomekFS

ZlomekFS is designed as distributed filesystem with maximal transparency of operations above it. It means that the operations should behave as similarly to operations performed above local filesystem as possible.

It means that such semantics as *immutable files* is not the best way to solve this problem. Of course there is a way how to make this semantics transparent to client application. It is briefly described in section 3.4 but there that way needs serious implementation optimizations to be usable in real environment. These optimizations are based on reusing data stored at physical storage.

stateless NFS protocol also provides interesting semantics. But in its pure form, it has only one image and all operations are serialized to that storage. When it is desired to some nodes have their own cache, it means conflicts can occur. Because the lack of open and close operations, synchronization operations must be done during other filesystem operations. But they do not define a consistent state of the filesystem. It is not always such a problem but restrictions based on its statelessness do not confirm to the requirement to be transparent as much as possible.

Unix semantics seems to be the best solution. But on the other hand, it requires a mutually exclusive mechanism to serialize operations. Some of these mechanisms are described here [19]. But all of them have one strong restriction. When the node wants to write, it must be connected typically to a significant part of all network. So the possibility of implementing this semantics mentioned here [20] is at least very restrictive.

It means that all nodes should be tightly connected and when a node wants to write to a file, it must obtain capability to write from a granting mechanism. After finished writing it should return the capability and simultaneously to returning the capability changes should be replicated through all network. This access does not solve synchronization of reading. But it is not a problem because *read own writes* semantics [21] at a node is granted by own operating system. And a read operation done concurrently to a write is not a problem because when such collision is done at local machine by two processes, the kernel always can schedule the reading process before the writing process.

The only problem would be the case when processes on different nodes perform network communication based on data read from filesystem. Then one process can read newer data than the other. At this case this semantics is not transparent. It is not hard to see the only solution of this problem is to succeed write only after successful installation of new data by all nodes. But this breaks one of the basic paradigms of distributed computing: There should not be done any expectations about speed or liability of a communication channel. But for data sharing between processes at one machine, it would be the fastest and the most transparent solution.

Last considered semantics is the *session semantics*. It mandates that only committed data are shown to other processes. Consider that the process in definition in section 3.5 is a node. It means there are special commit operations at that node, which shows internal changes to other nodes.

These operations should be the close actions of each process launched at node. It is because it is a way to say that the process has done some amount of work and has finished for now.

This feature brings the possibility of operation conflicts when writes are not visible to other nodes immediately. On the first sight, it could be a problem, but on the second sight, this property allows to replicate changes gradually through all hierarchy of network. It is because this semantics allows concurrent (in the meaning of parallel) writes on nodes (further discussion in 3.7) and no serializing authority is then needed.

So finally the conclusion is: for one node data sharing should be *Unix semantics* implemented and for communication between nodes the *session semantics* will be used. In the [1] has also been done discussion above choosing, but it didn't determine how the semantics would be used. And finally the real implementation of ZlomekFS didn't support any semantics in its pure shape. It is really true that data reintegration to server node was performed at the file close. But this is only one part of correct implementation of *session semantics*. When a file has been read, the local daemon has tried to update it from its server. It means that the opened file has been updated during read operation to a newer version than the version that was actually opened. Additionally, the update has been performed only by comparing data directly placed in the server's underlying filesystem. Again these data has not been committed by close operation. It has been only raw written data. During such update the image at server also could be changed by another subsequent write performed at server. This action has led to update interruption, but already updated data has not been rolled back to a really existing version.

A very similar situation is reintegration. When user process has reintegrated data to server, there has not been any mechanism which would prevent rewriting just reintegrated data. But what is worse: this way could create a file containing more than one version of data when one process has reintegrated data and the other process has modified another part of file.

In the end it is important to notice that this semantics is usable for *regular files* [1]. For other types of files, another semantics should be applied. For example named pipes or special devices should propagate all changes as soon as possible to other nodes.

3.7 Causality in ZlomekFS

We define causal dependency this way: If exist process p where action b depends on action a, written $a \rightarrow^{p} b$, then b causally depends on a ($a \rightarrow b$). For each message m $send(m) \rightarrow receive(m)$ and if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. This definition works well in an environment where actions are done atomically at one time.

But in a filesystem like the ZlomekFS, there rises a problem. The operation of modifying file could take quite a long time. It is bounded by file open and file close. As described above, on file close synchronization between nodes is done. But synchronization information can be delivered to a node with three possible states of the file. The first is: the file is closed. It means no other process is using that file. If the version of the file allows reintegration, the file is reintegrated, otherwise [1] a conflict is created at client node.

The second case: the file is just open only for reading. If the version of server's file is in conflict to the version of the client file, again conflict is created. But when the versions do not prohibit reintegration, there is a problem. Because of chosen semantics, the file can not be blindly rewritten. When the file would be treated in such a way, processes reading this file would have access to version of data which is newer than their opening and it s forbidden. Next, the conflict can not be created because there is no conflict in versions of the file. So the readers should obtain their own copy of the file and the file should be reintegrated. Then any new reader will open the reintegrated version.

The last case is reintegrating to file which is open for modifications. The correct behaviour is not easy to determine. When the file has already been modified, the answer is easy: conflict. But the question is how to behave when the file is opened for writing and has not been written yet.

When updating such a file, it means the opened version would be rewritten without any notification. In addition, there should be a mechanism which counts writes performed since the file opening. In addition, when a process tries to open a file for writing, it can be assumed that it will write the file soon. Because of these reasons, the write operation is not only the event when the physical write with version change occurs but it is the time interval between file open and close. So when a file already opened for writing is reintegrated, the conflict should be created at client.

But there is another problem with reintegration: when a reintegration is performed, the data could be sent to server in many chunks. This means that the reintegration is not an atomic operation. So between uploading the first and the last chunk of data any other process can access the file. It means that invalid mixture of versions would be accessed. This behaviour is also undesired, so all data (including versions *from* version and *to* version) sent during file reintegration are recorded to a journal and nothing else is done. When the client finishes the data upload, the file is locked. Then the *from* version is read from journal and compared to local version of that file. If they agree, the journal is traversed and all its content is merged with the file. When the versions do not agree, the journal is discarded and the client is notified about a conflict. In the end, the journal is removed and the file is unlocked. When, for any reason, the connection between client and server is aborted and other process tries to reintegrate same file, the journal is removed and a new one is created.

A very similar case is updating file from a client to a server. It should be performed whenever a process wants to open a file. When the file is not opened at that moment, the behaviour is clear. The file is updated and the new version is opened.

When the file is opened only for reading, the reading processes should gain their own copy and update can be performed. But problem arises, when there is a process owning a capability allowing writing. According to semantics of reintegration it means that the local copy of the file is in conflict with the remote. So the conflict is created. But the question is which copy should be provided to a newly calling process. The blind following of semantics says the remote, newer copy but here is other important fact. The file could be used to data sharing between two or more processes, so opening of the remote link to the file would affect this important facility. So the link to remote file is added to conflict directory but the local version of the file is provided to callee.

Another issue is tightly coupled to the strict following of session semantics. It rises when the data source for update is required by another node. When another node requires update, the hash result is counted from data stored directly at filesystem. When those data has been rewritten by any process and the process has not closed the file yet, the chosen sharing semantics does not permit to show the data to any other process.

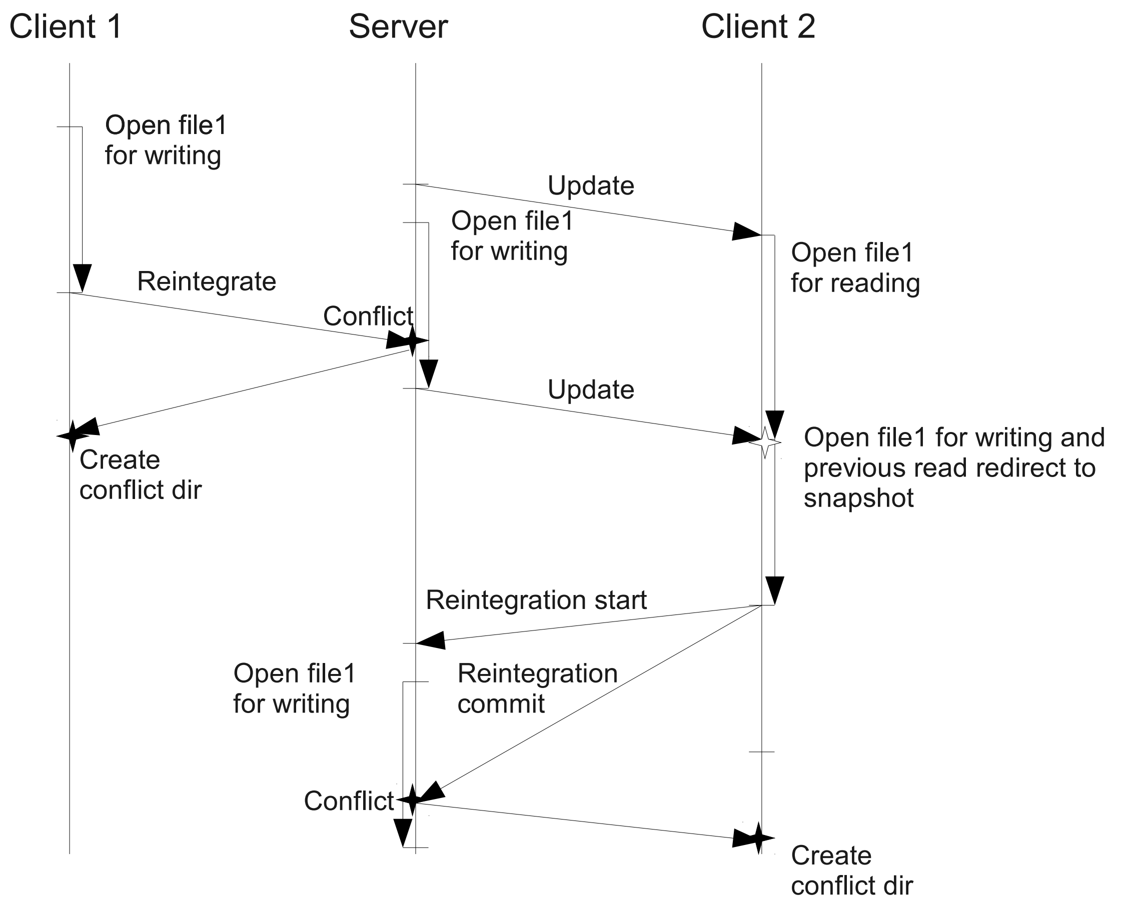

Figure 3.1: Reintegration ZFS semantics

This means another snapshot of such a file should be created when an open request is done to the filesystem (or at close operation). Then when an update request is performed, the data should be always read from the own cache, when no write capability is present. In other cases, a snapshot should be available with correct data and will be then used as data source of data.

For better understanding of this implementation of this semantics, a brief conclusion of reintegration is done in Figure 3.1 (for update is similar).

3.8 Solutions

As described above, there are some required changes to the ZlomekFS.

3.8.1 Reintegration/update journal

Because the reintegration or update are not atomic operations, as described above, there should be a way, how to make them atomic. Section 3.7 describes how the journal should work at all. But there are issues which should be solved.

The first is the place where the cached or such an action would be stored. There are several possibilities. The first is creating an unnamed temporary file. But this solution would waste system resources because when the reintegration is in progress, the file should not be closed. Because when such a file is closed, it is also removed from the filesystem [3]. Then another possibility where to place this temporary file is `/tmp` directory. But introducing this solution is another platform specific property of the project. So the best way is probably to be consistent with *directory journal* and place the *file journal* again to the metadata directory. This directory is not shared through the ZlomekFS infrastructure, so this could be a good choice. In addition the old implementation is able to build unique paths for files stored here dependent up to file's i-node number and device where the image of the file is stored. To recognize this metadata file from the others, the suffix must differ from other suffixes. In addition, there must be a way how to recognize update and reintegration type of journal because they can occur simultaneously. So the suffixes are *.update_file_journal* and *.reintegrate_file_journal*.

Another question to answer is the internal format of such a file and its precise behaviour when update or reintegration abort occurs. The main problem is whether these files can have the same internal format. First, each case should have as first entry metadata describing the source and the destination version of the file. This entry would be when the changes are commit whether the target file has not changed during operation. Then reintegration/update blocks themselves? should follow. The actual reintegration/update data has the same shape. It is always a block of data with specified length and offset where it should be placed. Because of this, these internal file formats can be the same, illustrated at Figure 3.2.

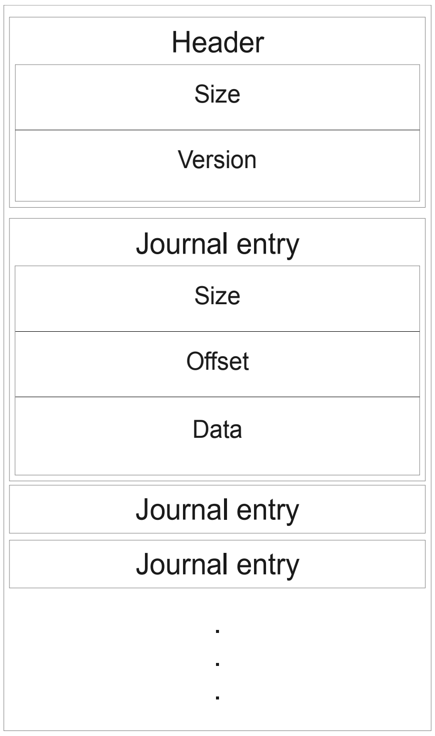

Figure 3.2: Journal format

But there are more issues to solve. When a reintegration occurs, it can be interrupted by some cases. For example losing network connection. Then the orphaned *journal* should be removed and the reintegration aborted. But the implementation of ZlomekFS does not allow to recognize the cause of disconnecting. The connection can be aborted because of a serious network error or by a fully intended action and the connection will be established again soon. Because of these reasons, there is no sensible way how to declare a reintegration as aborted. But when another reintegration from a node occurs and the previous node is disconnected, it can be assumed that an error has occurred. So the reintegration privilege is granted to a newly acquiring node and the *journal* is removed and started anew.

Of course, another case can occur. Suppose that one node is performing reintegration and has granted the reintegration privilege. Then another node wants to perform conflicting reintegration. There could be three basic ways how to solve this.

The first is to abort the previous reintegration. This means that there could occur a *starvation problem*. It means that many nodes would try to reintegrate and they would steal the right from others. The second way is waiting for the result of previous reintegration. When its result is conflicting to the waiting, the conflict would be created. But this solution would cause a problem when the first reintegration runs for a long time.

So there has been chosen the third way. When a node is trying to reintegrate a file already being reintegrated by another node, it probably means a conflict. It is because the reintegration changes the version of data stored in the file. And if the version of data of the remote version of the remote file corresponds to the local version and the remote file is being reintegrated, after commit the versions will not correspond. So in this case a conflict is created.

Another question is whether the reintegrated file should be locked at server. If yes, there is no possibility to change it but the reintegration can take a very long time, so it would case problems at that node. In addition, because of the issue described above: there is no way how to recognize failed reintegration, the file could be locked forever. So the file should be locked only when the *journal* replay is performed. The update operation is more less the same, accordingly to section 3.7.

3.8.2 Snapshots

As mentioned above, there must be a way how to store snapshots of files. The two basic ways could be used. The first is storing all the file how it lies in the filesystem. But this solution wastes the place on the persistent storage and, in addition, such an action would be very slow. Because copying file means its reading, and again writing all its content to the disk.

The results of [3] seemed very interesting. But in fact, it can not be used for implementing *Unix semantics* because it forbids reads parallel to writes. Access to files is treated in a special way. When a user tries to open a file for writing but the snapshot already exists, the writing operation is rejected. But the idea of storing differential intervals of files could be a good idea. So the solution of creating snapshots is quite similar to the implementation of version system. As a result, it should not be exclusive to version system. These two accesses to treating snapshots should work together.

The idea is very simple. Every time when a special event occurs, a new snapshot is created. The snapshot is an empty file joined to an interval tree. Whenever the file is then modified, the modified interval is backed up and its boundaries are stored in the interval tree. When the old version of data is then requested, the current version of file is read and then the interval tree is traversed. If there is an intersection between read interval and data already backed up, the data is “updated” from the copy of the old version.

Another interesting case is when the source file is then truncated. It means that all data contained in the truncated area must be stored to backup area. This is very similar to the [3].

The important question is when the snapshots should be created. As described above, such events happen when local data is modified by a reintegration or update (the commit of such an operation). It has been discussed that no writing capability should be present at the system at such an event. Then all read-only capabilities should get assigned that snapshot and correct its readings by data of the right version.

But another event for creating a snapshot is required. Because when an update or reintegration is performed, then the data source should be a snapshot. In other case the data of an incorrect version could be transferred. For reintegration it is not a problem because it is performed only on file close. But when an update is performed, it should be done above a version snapshot. As described above, the events when a snapshot for update should be done, are file closes.

When later a data update occurs, all operations are done above a snapshot. At the end of update file attributes should be also synchronized. It means the snapshot should also contain metadata of the file. And in addition, there is a question how to treat fisle attributes changes: whether the file attribute change is a part of the sequence of operations, or it is a standalone operation which is able to start a new snapshot of file.

The operation of file's attributes change can be understood in the same way as a commit of the current file state. So this operation would start a new snapshot for update of the file. On the other hand, this type of access would bring a new type of conflict: attribute-modify where one process changes attributes and the second data inside of the file.

But another issue rises from creating snapshots. There can be a process, which reads from a snapshot for a very long time and so the snapshot has been overlaid by many other updates and snapshots. In addition, there should be a way how to store many version snapshots. One possibility is to have a complete history of changes for each version used by processes. But when this way is used, it has many disadvantages: first, every change to current version of file triggers many data copies at underlying filesystem. It would mean a significant performance loss. The second is then significant disk space requirements.

Here can be done an observation: let have snapshots a and b where $version(a) < version(b)$ then all changes needed to reconstruct b from the current version are subset of changes stored in the same way to reconstruct snapshot a . In result the solution is quite easy. Whenever a new snapshot is created, then any older snapshot is closed and stored until any capability aiming to its version exists.

When a read operation to a snapshot occurs, then all newer snapshots are applied in the order they were created. Last, the snapshot of a specified version is applied and the read data should be in the correct shape.

From previous flows, a good way of storage of snapshots is a chain, which could be traversed when searching the correct version of the snapshot of the file. The schema of creating snapshots is shown at Figure 3.3.

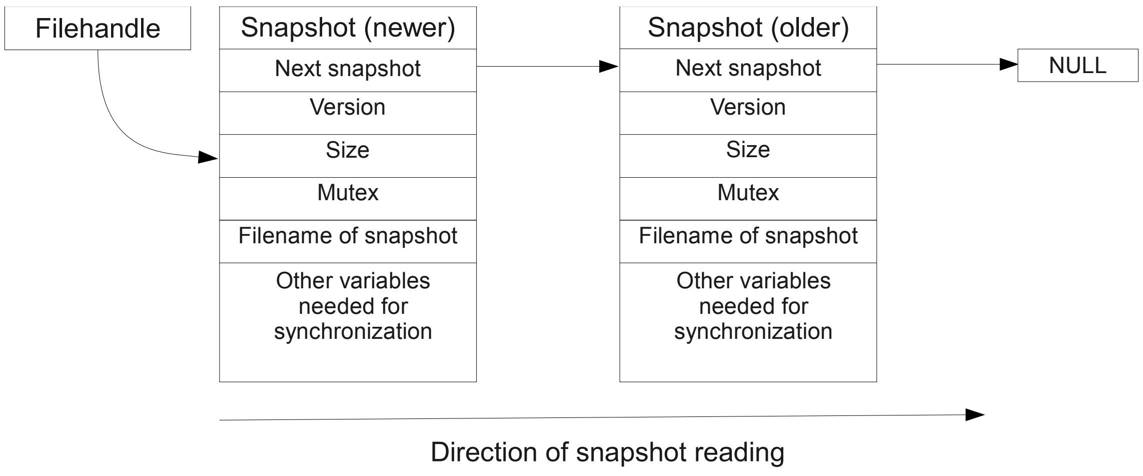

Figure 3.3: Snapshot handling

But a little problem occurs: when a snapshot for reintegration or update is created, it would cause transferring all current read-only capabilities of such a file to a reading snapshot. But it would break the property of *Unix semantics* at a local node. So another flags were introduced: `O_RUPD` and `O_RREI`. When a file is opened In such a way, the new capability is created and the latest snapshot of such a file is assigned to it. This is a safe way how to ensure that all operations would be done above a snapshot and no other processes will be affected by this behaviour.

4 Capabilities and filehandles

As mentioned above, capabilities are used to determine which actions can be done to the opened file. But when capabilities are stored at remote nodes, there is no way how to change data contained. Even capabilities stored at a local node are hard to change. And as described above, there should be a way how to change them when a snapshot is created. It is because the version of snapshot which is tied to the capability should be stored in the capability. So there should be a mechanism which allows to change a capability that is not present at the kernel of the ZlomekFS.

4.1 Remote capabilities

There are two basic ways: to notify the client holding the capability, to change it according to the request. But this solution brings a necessity of network protocol changes. In addition, it would be necessary to send this information in direction from server to client. And with the old implementation of ZlomekFS it would be a serious amount of work.

A better solution is to publish only virtual identifiers of filehandles. It could be only a large integer number followed by a bit-field which contains the access rights and finished with a field with a digest, which proves the validity of the capability.

When such a filehandle is then passed to the ZlomekFS core, first the digest is checked and if it does not seem to be valid, the operation is rejected. In other case, the capability identifier is then used to translate this handle to real capability which is then used to perform an operation.

4.2 Capability securing

The capability format, especially the level of securing, as described at [1] works well. It is because of the known and trusted environment. But in the environment where an attacker can impersonate a trusted user machine, there should be a way how to revocate an issued capability.

These revocations should be done on file close. But when another process has opened the file for the same access right, the capability is shared among the processes. This means that just one close does not remove the capability from the system, so another access via already closed file can be performed. It would be a problem in case when the file has been closed and its access rights has been changed. Because of this each open operation should get its own capability which will be revoked at the corresponding close.

Even these requests are handled by the mechanism of remote capabilities. The underlying core of system can henceforward use the old mechanism of capabilities. But requests performed from the outside of the secure core of the filesystem have to prove their legitimacy by remote capabilities which are unique per open.

Therefore there should be a mechanism for signing the remote capabilities in order to check their correctness. The old model of copying the random number hash from the capability holder is not sufficient. It is because the `MD5` has been used to count the verify. This algorithm is very useful for checking data for a random change but is not resistant to an aimed attack [22]. Because of this an attacker could get the state of the internal pseudo-random generator. A better solution would be signing capabilities by `SHA256` or newer algorithm. But signing itself is not sufficient. The result should be a field of unique salt created by hash mentioned above and a real hash calculated from the content of remote capability and the salt. When the hash is calculated this way, the remote capability is valid even when its issuer has restarted, but its image does not exist. It is a problem that can be solved in two ways.

The first solution is a persistent storage of capabilities but it means a serious change of ZlomekFS daemon architecture. A better way is to add an instance specific value (random number) which would be also an input parameter of the hash. Then after restart of the daemon, all capabilities will be invalidated. When a client acquires an operation with such a capability, the operation is rejected with code `CAP_OBSOLETE`. Then it is up to the client how to manage this case. Whether to fail the operation or try to reopen the resource and thus get a new capability.

4.3 Capability ageing

Issuing capabilities also brings problems with their storing. Each capability takes some amount of memory. So when too many capabilities are present in a system, it would lead to wasting memory. Here are two basic possible approaches. One way used for example at [23] is to remember for how long an open file has not been accessed. And when the time goes over some boundary, the capability is discarded.

But ZlomekFS is designed to emulate local filesystem remotely and this solution is not transparent. Because at local filesystem, the file can be opened for any time and it can not be closed by the operating system. To be most precise, it would need a deep analysis of times when the files are left untouched.

Then a sensible boundary should be set and a mechanism how to remove capabilities considered as unused should be implemented.

4.4 Filehandle

As described at [3], there is a performance problem tied to filehandles implementation. The filehandles contain i-node number, which is specific for one instance of a file. But when a file is truncated its previous version should be stored and the simplest and fastest way is to move the image of the previous version to the version storage and then create a new empty file in the local cache. But the newly created file will obtain a new i-node number and this operation will invalidate all filehandles held by other nodes aiming to such a file.

In addition, when a file is physically deleted from the filesystem, the i-node is then again available for a new file. It rises a very serious issue. When a filehandle containing the i-node number is somehow cached by FUSE interface (or a client), then a problem could occur.

Suppose one process has requested a filehandle which is then stored at the interlayer between FUSE and ZlomekFS. Then an unlink operation is invoked. It successfully removes the file from the filesystem. So the i-node number becomes available to a new file. At this moment a file creation can be performed and the file could get the same i-node number. Unfortunately such file can be a file created by another user and somewhere in the filesystem.

After these actions the first process holding already invalidated filehandle can try to open its properly gained filehandle. The i-node number agrees with the newly created file, so it is found in internal structures and the file is opened.

This issue could be solved by introducing generations of filehandles according to [1]. For better illustration the scheme of the filehandle is shown in Figure 4.1. But this solution is quite complicated and not very well extensible. A better solution would be assigning a unique virtual integer identifier to each file – internal ZFS i-node number.

Filehandle
ZFS file identificator
Flags
Version
Serial number
Filename of snapshot
Signs

Figure 4.1: Filehandle

This will allow to change the physical image of file without any affect to the filehandle. But even this solution is not without any problems. Many operations are done in such a way where the i-node number of file in cache is got by calling `lstat` call and then a partial filehandle is passed to `lookup_metadata`, which finishes the creation of the filehandle.

It can be seen that all ZlomekFS is internally deeply tight to i-node numbers of the underlying filesystem. To satisfy such requirements of the ZlomekFS, there must be implemented an interlayer which is able to assign a unique integer to each newly created filesystem entity and for all known old i-nodes will remember the mapping of the underlying filesystem i-node number to the virtual i-node number of ZlomekFS.

4.4.1 Unique integer generator

So the solution is made from two basic parts: the generator of unique identifiers and the mapping interlayer. The question is how the generator should be designed. Suppose the case where a client is tightly coupled with his server. It is the case of current implementation of ZlomekFS where a filehandle to a remote file is just a local filehandle of its server. In such case the generator can be only an atomic sequence number generator. On the other hand, the last number sequence should be stored in a persistent storage. It is caused by the necessity of having for each file machine-wide unique identifier and after the restart of the daemon, the sequence must not be repeated.

So because of this, the sequence generator is designed as 64-bit wide unsigned integer which is shadowed to a persistent storage. So the question is where the integer should be stored. The filehandle also contains volume identifier which is used to determine which volume contains the represented file. So for such a file the local storage of volume metadata can be chosen.

But when each file creation would generate an additional I/O request to local filesystem, it would mean an unnecessary significant performance issue. So the sequence generator should be represented by an object, which acquires an interval of values from the persistent storage. Then it assigns them just by in-memory operations and only when the interval is exhausted, it acquires another one.

4.4.2 I-node numbers mapping

However the described solution is just a part of a more complex system. Because of the issues described above, there must be present a system which does mapping from physical i-node numbers to the virtual i-node numbers. This can be done by reusing already implemented structure: `hash_file`. Whenever a new file is in the filesystem created, a virtual i-node number is acquired. Then a pair which is inserted to the `hash_file` is created from physical and virtual i-node numbers.

Then when any ZlomekFS procedure requires a filehandle from a physical i-node number, it asks for the virtual one. When it already exists, the mapping is found and the filehandle is ready for use. In case when mapping for such number does not exist, it is created and such a file can be from now accessed by its physical i-node number.

5 Minor changes

Except large changes described in previous parts related to data sharing and network security, there were made many minor changes to ZlomekFS daemon to improve its stability.

The biggest problem of the whole implementation is the threading model. It is designed to allow massive parallelism, but because many levels of locking and unlocking, it leads to race conditions. Overall it is the source of most of stability problems. So some changes to locking schema were made with the aim to improve the stability but when a change has been made, another issue has raised. Many times the new issue did not relate to the new change. Just another race condition manifested.

But some minor changes have been made. The calling `mlock` procedure was omitted – in modern system it is not necessary and for launching the daemon under the account of a regular user it is unusable.

Next the problem when the node is connecting to itself has been documented. When this situation occurs, a deadlock is created and all daemon is blocked.

By storing access rights in metadata there has been solved a serious bug in file opening. It occurred when a user created file with write only privileges and then the daemon tried to open it (it opens files with read/write access) the opening failed – now all files at local cache are writeable by the owner of the daemon and belong directly him.

Another serious bug was wrong reintegration, when the file on the server was only truncated and the reintegration finished. This was caused by wrong versioning of file, which has been fixed during implementation of transact reintegration.

6 User manual

The first step is to obtain source codes from the SVN repository placed at https://shiva.ms.mff.cuni.cz/svn/zlomekfs/branches/zaloha/stable and to install necessary libraries to build the filesystem. First, the script `$./makeall.sh` from the root of tree of source code should be launched. It will build all parts of the daemon. If this step is successfully passed, it is necessary to launch `#make install`. This will install all parts of ZlomekFS and create user *zfsuser* and group *zfsgroup* to securely launch the daemon. During creating the prompt to enter *zfsuser* password is displayed. It is good to remember this password to administrate ZlomekFS without root privileges.

After succeeding this basic part of installation, the daemon requires special setting up. Firstly, setting up and launching daemon for a machine is described.

It is necessary to set-up all settings required by previous installation described in [1]. But the SSL framework requires some other information to successful work. First, it is necessary to generate a pair of a root key and a certificate (if it already exists, it is necessary to convert the certificate to the PEM format and generate all user keys and certificates in the PEM format). This can be done by launching the script `CA.sh` placed in the directory `ssl_utils` in the root of checked out repository. The pair contains two files – the key and the certificate with same prefix. This prefix is passed to the script as a parameter:

```
$./CA.sh prefix
```

If the certification authority is created, it is necessary to create a machine certificate. This action should be done by script `machine.sh` placed again in the directory `ssl_utils`, the first parameter is the prefix of the name of a newly created pair of a key and a certificate, and the second parameter is the prefix of the name of the certification authority (it should be placed at the same directory from where is the script launched):

```
$./machine.sh new_pair ca_prefix
```

The Creating of this pair is necessary for each machine added to the ZlomekFS infrastructure. When these certificates are created, DH parameters and a random seed should be generated. It should be done by launching special `gen_params.sh` script in the `ssl_utils` directory. Log in as the *zfsuser* and generate DH parameters with length 512, 1024, 2048 bits in files `dh512.pem`, `dh1024.pem` and `dh2048.pem` and random seed file `rand.pem`:

```
$gen_params.sh
```

Please note that while creating these files, some other files are created, so do not delete or modify them and do not allow any other users to access the directory where they are stored.

Next for all generated files the owner should be changed to *zfsuser*, the group to *zfsgroup* and their `mode` to `600` before installing them to their proper places (eg. `/etc/zfs`). The installation is then done by simple copying them to the destination. After having completed it, you should change paths aiming to them in the `/etc/zfs/ssl_config`. Necessary values are `Localcert` – path to the machine's certificate, `Localkey` – path to the machine's key, `Verifycert` – path to the certification authority certificate, `Password` – password used to decrypt the local key, `dh[512|1024|2048]` – path to the DH parameter files and `seed` – path to the random number seed. Remember, the paths should be accessible for the *zfsuser.*

When these properties are set, it is necessary to create the certificate revocation list (CRL). If built in scripts for generating certificates were used, it should done by calling in the directory where the scripts are placed :

```
$openssl ca -gencrl -keyfile [ca_prefix]key.pem -cert
[ca_prefix]cert.pem -out revocated.pem -conf Caconf.con
```

The file `revocated.pem` should be then placed to the root of configuration volume.

Finally, the group membership file should be edited accordingly to the situation. It should contain at least one line. Its internal format is line:

```
user_id: group_id[, group_id[, group_id]...]
```

Where all elements are integer numbers of ZlomekFS internal users and groups. When an access right check for a user type client is performed, the group membership of this user is read from this file.

Now the daemon can be launched as root and as parameter should be passed the mountpoint where the infrastrucutre should be placed. When it starts, its access rights are decreased to the *zfsuser*:

```
#zfsd mounpoint
```

The last important thing is generating a new version of the CRL. It is done again by openssl framework. If the built-in scripts are used to generate certificates, it should be done by calling in directory where the scripts are placed:

```
$openssl ca -revoke [bad_prefix]cert.pem -keyfile
[ca_prefix]key.pem -cert [ca_prefix]cert.pem -config
Caconf.conf
```

When the CRL is updated, it should be signed by the same way as the first CRL was generated and placed to the root of config volume.

The setting up of user machine type is quite similar. But because of launching the daemon in a jail, it has some specifics. First, the certificate for the client should be issued. It should be done by calling

```
$./user.sh new_pair ca_prefix
```

It will create a new pair of a certificate and a key. During the creation you will be asked to enter a proper ZlomekFS user id of the owner of the certificate. It is necessary to fill it correctly because it will be used for access rights checks.

Then a jail should be created. It is a directory where the daemon will be placed. It should be in a directory where the user launching the ZlomekFS has write access and the *ZFS-user* has at least read access. In this directory directories `var` and `etc` should be created. In the `var` the local cache of the created volume should be placed. Then the image of the `/etc/zfs` directory (including all content needed for the security features) should be placed somewhere.

Then should be called:

```
$zfsd_launcher config_dir mountpoint
```

Where `config_dir` is the image of `/etc/zfs` and `mountpoint` is the mountpoint with absolute path (begins with `/`) from the root of the jail. After this, the ZlomekFS infrastructure should be created and mounted.

7 Conclusion

The goals of this thesis have been mostly met. The security model has been fully implemented and seems to work well. The standard network security framework – OpenSSL has been used. It brings very high security of data transfers and it is quite user friendly to an administrator.

It is now possible to logically divide the ZlomekFS infrastructure to fully trusted machines which contain all data, and single-user clients who are allowed to access only their working set of data. The resolving of their types is based on signed certificates, so it should be secure. In addition, there has been added feature of kernel check permissions, so now any user should not have access to data which doesn't belong to him.

This thesis also considers the case when user's certificate can be compromised, so the possibility of revocation obsolete certificates has been added.

The whole implementation of the network security framework has been done by an abstract layer, which hides implementation details from the rest of the system. Because of this, another implementation of network security model can be done without large changes of the current source code.

Fixing the file sharing semantics has been implemented. Not only an easy session semantics fix has been implemented, but also new distributed file sharing semantics has been defined and implemented. This new semantics differs from the classic session semantics which says that each opened file descriptor should have access to the version of file existing at the moment it has been opened. But when concurrent writes at one machine occur, the session semantics rule is broken and the second opening of the file should fail.

But the new implementation works with local files in the same way as *Unix semantics* until they are changed remotely. Then if the file is not opened for writing, and versions allow to do the change, it is done. Otherwise, in this semantics, it means a conflict has occurred and it is created.

A relatively easy but very important fix of abstracting filehandles from underlying i-node numbers has been done. This fix is realized by assigning unique virtual integer identifiers to the i-node numbers obtained from the underlying filesystem. This patch allows performance improvement related to versioning, as described in [3].

The last goal – making the ZFS daemon stable has not been fully satisfied. Even more than one thousand hours spent by trying to fix all assertion aborts did not lead to success. It is because the very concurrent architecture of the daemon and quite confusing way of implementation where there are many levels of locking entities and operations are not fully unified to require the same level of locking. In addition, many functions called inside the daemon as side effect unlock some locks dependently on satisfied conditions. So fixing the stability issues of ZlomekFS should joined with overall change of design of the application.

7.1 Further work

As described above, future work on ZlomekFS should aim at changing the locking model of all the system, unifying the requirements of locked mutexes of all levels of abstraction. Probably a good way to achieve this goal is to make the daemon less parallel and to reduce the amount of mutexes and threads. On the other hand, the main ideas of the architecture of the daemon are very interesting.

Another required fix is implementation of access rights check for directory tree. At this moment it is implementable only for files cached locally. So when a node without cache is placed between fully cached node and leaf node with user level certificate, there is a problem: The node without cache can not check access rights of the path to the file and it can result in access rights policy violation.

Bibliography

[1] *Shared File System for a Cluster* [online]. Prague, 2004 [cit. 2012-04-10]. Available at: https://shiva.ms.mff.cuni.cz/svn/zzzzzfs/trunk/doc/Zlomek-SharedFileSystem.pdf. Master thesis. Faculty of Mathematics and Physics.

[2] RFC 1094. *NFS: Network File System Protocol Specification*. 2550 Garcia Avenue, Mountain View, CA 94043, 1989. Available at http://tools.ietf.org/html/rfc1094

[3] WARTIAK, Rastislav. *History and Backup Support for ZlomekFS* [online]. Prague, 2010 [cit. 2012-04-10]. Available at: https://shiva.ms.mff.cuni.cz/svn/zlomekfs/trunk/doc/Wartiak-HistoryBackup.pdf. Master thesis. Faculty of Mathematics and Physics.

[4] RFC 791. *DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION*. 1400 Wilson Boulevard, Arlington, Virginia 22209, 1981. Available at: http://www.ietf.org/rfc/rfc791.txt

[5] RFC 2460. *Internet Protocol, Version 6 (IPv6) Specification*. 1998. Available at: http://tools.ietf.org/html/rfc2460

[6] *Truecrypt* [online]. 12 [cit. 2012-04-10]. Available at: http://www.truecrypt.org/

[7] *Dm-crypt* [online]. [cit. 2012-04-10]. Available at: http://www.saout.de/misc/dm-crypt/

[8] RW PENNEY. *Cryptmount* [online]. 30.3.2012 [cit. 2012-04-10]. Available at: http://cryptmount.sourceforge.net/

[9] Lest We Remember: Cold Boot Attacks on Encryption Keys. *Proc. 2008 USENIX Security Symposium* [online]. 2008[cit. 2012-04-10]. Available at: http://citp.princeton.edu.nyud.net/pub/coldboot.pdf

[10] KAUFMAN, Charlie, Radia PERLMAN a Michael SPECINER. *Network security*: *private communication in a public world*. Englewood Cliffs, New Jersey: PTR Prentice Hall, c1995, 504 s. ISBN 01-306-1466-1.

[11] CERT/CC Denial of Service Attacks. CARNEGIE MELLON UNIVERSITY. *Http://www.cert.org* [online]. [cit. 2012-04-10]. Available at: http://www.cert.org/tech_tips/denial_of_service.html

[12] ADAMS, Carlisle, Steve LLOYD a Carlisle ADAMS. *Understanding PKI*: *concepts, standards, and deployment considerations*. 2nd ed. Boston: Addison-Wesley, 2003, 322 s. ISBN 06-723-2391-5.

[13] VIEGA, John, Matt MESSIER a Pravir CHANDRA. *Network security with OpenSSL*. 1st ed. Sebastopol, CA: O'Reilly, c2002, 367 s. ISBN 05-960-0270-X.

[14] *Linux man pages* [online]. [cit. 2012-04-10]. Available at: http://linux.die.net/man/

[15] *FUSE* [online]. [cit. 2012-04-10]. Available at: http://fuse.sf.net

[16] How to break out of a chroot() jail. *Http://www.bpfh.net* [online]. 12.05.2002 [cit. 2012-04-10]. Available at: http://www.bpfh.net/simes/computing/chroot-break.html

[17] BUTENHOF, D. *Programming with POSIX threads*. Boston: Addison-Wesley, 1997, 381 s. ISBN 02-016-3392-2.

[18] TANEBAUM, Andrew S., KAASHOEK, Robbert VAN RENESSE a Henri E. BAL. *The Amoeba distributed operating system*: *Status report* [online]. Amsterdam, Netherlands, 20.03.1994 [cit. 2012-04-10]. Available at: www.cs.vu.nl/pub/papers/amoeba/comcom91.ps.Z. Paper. Vrije Universiteit.

[19] TANENBAUM, Andrew S. *Distributed operating systems*. New Jersey: Prentice-Hall, 1995, 614 s. ISBN 01-321-9908-4.

[20] *OpenAFS* [online]. [cit. 2012-04-10]. Available at: http://www.openafs.org

[21] ZAVORAL, Filip. *Distribuované operační systémy.* The textbook for the course of the same name, Faculty of Mathematics and Physics.

[22] WANG, Xiaoyun a Hongbo YU. *How to Break MD5 and Other Hash Functions*. [online]. 200[cit. 2012-04-10]. Available at: merlot.usc.edu/csac-f06/papers/Wang05a.pdf

[23] *Coda File System* [online]. [cit. 2012-04-10]. Available at: http://www.coda.cs.cmu.edu/

A Enclosed CD