UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science
Specialty of Information Technology

Dmitri Borissenko

Integrative Graph File Systems

Master Thesis (30 ECP)

Supervisor: Ulrich Norbisrath

Author: .....”.....“ June 2011

Supervisor: .....”.....“ June 2011

Allow to Defense:
Professor ...”.....“June 2011

TARTU 2011

Contents

Acknowledgments

There are some people I would like to thank for their help in writing this thesis. First of all, I would like to thank my supervisor Ulrich Norbisrath. He offered many ideas about the thesis and helped me a great deal with many things, starting with Python and Linux basics and finishing with conflicts around the conceptual ideas of the thesis. I had many interesting discussions with Ulrich regarding the elicitation of requirements and solving some inconsistencies with work concepts. I would also like to thank Dmitri Danilov for the explanations about his work (Grapth3D) and for qualified help with integration. Also, I would like to thank my parents, Lidia and Aleksey Borissenko, who gave me the required mental support. A special thanks to Artjom Lind, who showed me some tricks in Linux.

Chapter 1

Introduction

This chapter introduces the thesis and explains the motivations behind it with the help of a sample scenario. Also, required knowledge and prerequisites are provided. A short definition of the main goals for the future will be given here as well.

1.1 Motivation

The motivation of the current thesis is information overload. Each person has his or her own unique collection of files: movies, various writings, presentations, images, photos, audio files, blueprints, letters, etc. After a given period of time the amount of data usually increases, and a simple operation could take much longer than expected. At the moment it is difficult to sort knowledge and make sense of it without technology. The huge variety of documentation covers many different subjects, and a single person can become quite disoriented. Many pieces of information can cover the same idea, with its unique deviation from the truth and errors inside. With the Internet, the ability of society to navigate and organize through this mess has been increased multiple times. Hypertext permits us to represent the information in a more natural way: the unified web of knowledge sources.

Our memory works through associations – and this is the primary key in organizing "semantic nets". Modern navigation and search systems allow for different methods of sorting information. Take the company Google, for example. They constructed a search engine[7] which has assumed the lead position on the Web for a long time of time. Grouping information by search type criteria allowed unnecessary pages to be separated from desired ones. A ranking system for sites yields significantly better results for a specific search. The main principle is pretty simple: the more attractive and popular a web-site is, the more ranking points and better index position it has in the global top. This is quite a simple and effective principle for qualifying information. Additional features of the Google search engine are search options. There are additional capacities for searching images, articles, mail, blogs, sites, and photos. Even a shopping engine is available. It is a significant advantage that people are able to qualify from the beginning the data they seek, and get back sorted and filtered results. The Google example proves the necessity and importance of the procedure of information pre-processing. If one company is able to earn good money by analyzing and sorting open data sources, perhaps it is a signal for other (not Web) domains to do something similar. For instance, public shared service of user space (e.g. the university), or even simple home data collection (such as photos, music, text-documents, etc.).

In order to better understand the priority and significance of the problem, lets classify by groups the typical structures of user's data organization methods, starting with the user at home. It could be not necessarily limited to a single computer (however, this is a major case in this group), but may be a small home-made network with a couple of interconnected machines acting as a single system. User data is stored on 2-3 local PCs and united into a single private home network. It is traditional to classify information in such systems using files and folders. So, the final user interacts only with hierarchical data structure. Going to the upper level, the next group could be the small business network of a small company or organization. Financial limitations do not allow them to significantly upgrade the maintenance of inner data by developing individual solutions. Therefore, the only solution is using existing ones (folders + files), which means again storing, classifying and interacting with information mostly in the hierarchical structure. The last group is composed of large-scale organizations, communities and companies. The means of data organization could vary here appreciably, starting from the small databases and interfaces for data access and finishing with the large complex mash-ups[13]. Most classifications and ways of organizing information imply some work-specific interfaces. So, the third group can take care of its data representation efficiency, the second one - slightly, while the first one actually has no such opportunity.

Coming back to the data organization efficiency problem, it might be useful to analyze the current state of the first and second groups mentioned in the previous paragraph. So, in most cases, people organize their data with folders and files inside. This means that the efficiency of data organizing could be optimized here. What are "file" and "folder" actually? From a low-level perspective both of these types are quite similar, except that the folder provides information about the files "inside" and the file holds content itself. Earlier versions of hierarchical file systems (HFS) used flat table structures[10]. Then these structures were replaced with the Catalog File, which uses a B-tree structure allowing for much faster searching. Currently, it is normal to see something like files listed in some folder, ordered by time of creation or name.

1.2 Sample scenario

Now let us consider the following scenario as an example:

Suppose that Ted is a Senior Researcher at the university. He also received a position as lecturer, which adds the responsibility of four active courses, and each course assumes two lectures per week plus two practical lessons. Each course contains 20-50 students (at the end of the semester the total number of active students usually decreases). Each semester lasts 16 weeks and this usually means that on at least 14 occasions there is a need to do special preparations (like updating old files, demos, and presentations). Usually, preparatory operations take a lot of time and about 30% is spent searching for similar materials from previous years. Additionally, Ted receives at least 50 e-mails per day, and every 10th e-mail is some student's homework solution. Since each year brings more new content, Ted wants to improve the efficiency of interacting with his data.

Problem # 1. So, what Ted usually does is continuously sorting and updating content. To do so, he splits data into separate categories: homework – all files from students; teaching – all related content (such as slides, images, demos); personal –

private files; research – some test results and sample prototypes of software. There are a lot of wide-spread file collections in a data tree, so sometimes it is very problematic for Ted to find a specific file, especially in the case of an old file (Ted tends to forget old file names).

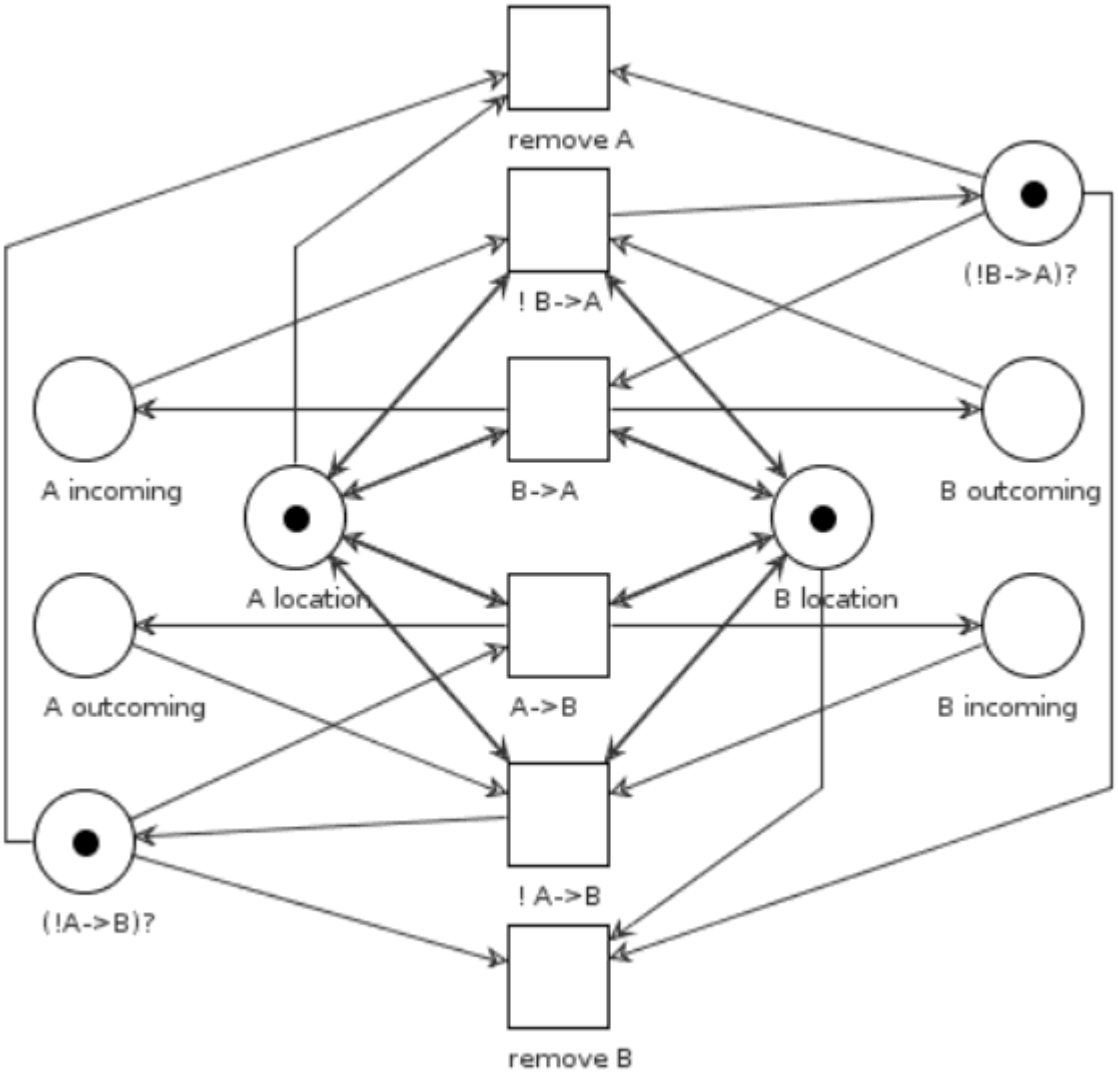
Problem # 2. It is often not enough to physically put data into separate directories. For example, some brilliantly done homework, which Ted uses as lecture material. In order to avoid file duplications, he creates links. However, when a particular file becomes obsolete, it takes a lot of time and effort to delete it completely. When Ted uses hard links, files remain in other locations (where the hard links were created). When using soft links, many broken links remain somewhere in a data tree. In the case of moving a target file to another location, the broken links problem is actual as well.

Problem # 3. After a few years of research work, Ted opens the "research" category and notes that approximately half of these files (on different levels of the data tree) can also be logically organized into three new subcategories: "prototypes", "utilities" and "manuals". However, it is currently impossible to do so, because current logical categorizations (by project) will be broken. In case of using links, Ted will be required to create hundreds of links, since these files are wide spread in the hierarchy.

1.3 Bidirectional linking.

The main problems of the scenario described in the previous section can be easily covered with one of Ted Nelson's ideas [**?**] about the Internet model, which assumes the presence of bidirectional relationships (bidirectional linking) between two objects. The idea is simple: if one object points to another one, then it has to receive a kind of confirmation or approval and only then such a reference will be legal. The Petri Nets [**?**] diagram 1.1 captures the possible states of three operations with links (link, unlink and remove a target) between A and B locations. These locations can be interpreted as some abstract points in any graph structure based system (e.g. two files in file systems or two web-pages). Bidirectional linking assumes parallel editing on both points, which causes many problems (such as synchronization, host reachability and network delays) if the destination point is located somewhere in the network. But, in general, if we deal exclusively with our own user space , then bidirectional linking becomes more actual, since most of our troubles (like starting a particular server on time, granting required access rights to a shared file) could be solved personally by a user.

It might be reasonable, before creating something more global and big, to make sure, that such systems with bidirectional linking can exist. One possible solution was found in the face of the FUSE platform[**?**](see chapter 3 for more details). It provides an extension platform for a variety of operating systems (OS) and allows for custom implementations of some basic file system utilities, such as rename, delete, create file/directory, link, unlink and others (see official documentation for complete list). The first tries showed that such systems could exist on a small scale. More serious approaches require more work-hours and more technologically advanced solutions.



A -> B: target A points to B
! B -> A: target B removes pointer to A
A incoming: references to A from other targets
A outcoming: all A pointers to other targets

Figure 1.1: sample bidirectional linking

Prerequisites

To better understand this thesis, knowledge about the following is recommended:

- [optional] Python[15]. The source code is implemented purely in Python. FUSE implementation is also in Python, but the original version of it can be found in C++ language[2]. Thus, this knowledge could possibly provide some answers to the technical questions.
- Graph structures[8] concept. The main work is done in graph structures. This is required for the ability to understand the work features.
- In the current work some file systems utilities will be mentioned. Reader should have a little hands-on experience with different types of file systems.
- [optional] Python supporting IDE[1] (e.g. Eclipse[3]). It is more likely for the source code to be viewed through a special interface (not just open the code as a text file). This will significantly increase the speed of navigating through the code, reading and compiling ability.
- Petri Nets. Some of examples are explained through the Petri Nets diagrams.

Goals

The main goals of this work are to prove the possibility of the existence of an integrative file system with bidirectional linking, and to demonstrate the advantages of such a feature. Integration means the possibility to use such a file system with the standard file operation and exploration tools available in current operating systems.

Outline

The thesis is organized as follows. Each chapter has a brief introduction of its content. The first chapter introduces the thesis and explains the motivations behind it. As an example, a possible scenario is provided. The key feature of the work "bidirectional linking" is briefly introduced. Also, a list with the requisite knowledge is provided along with the main goals of the work. The second and third chapters are mostly about related work. In these chapters, an analysis of existing tools is provided. Also some of the initial opinions of experienced people are discussed. Finally, the file system in user space will be described as a part of related work. The fourth chapter covers the requirements related issues. The fifth and final chapter gives a detailed explanation about the implemented file system, its possible disadvantages and advantages, tests and future works.

[1] IDE - integrated development environment.

Chapter 2

Related work

In this chapter, related works are described. First of all, the existing features are analyzed using examples. This is followed by an analysis of some discussions by people regarding problems that will be performed. Finally, an introduction to tagging is covered.

2.1 Existing features

So, what features/utilities are available now and do they cover actual needs? It was already mentioned at the end of the previous paragraph that files can be sorted by properties. Lets take a closer look at the Unix command [23] command "ls"[22] as an example. In listing 2.1 the fragment of "ls -l" output is provided with several properties of files. These properties are:

1. File type
2. Permissions
3. Number of hard links (we will cover hard links later)
4. Owner
5. Group
6. Size
7. Date
8. File name

```
>ls -l Pictures
-rw-r--r-- 1 user user      125 2010-11-13 03:48 alliance.gif
drwxr-xr-x 2 user user     4096 2011-03-06 19:35 data
-rw-r--r-- 1 user user      871 2010-11-13 04:36 sample2.png
-rw-r--r-- 1 user user     9359 2010-11-13 03:46 sample.odg
-rw-r--r-- 1 user user   104801 2010-11-13 04:39 Screen.jpg
-rw-r--r-- 1 user user   105017 2010-11-13 04:38 Screen-1.png
-rw-r--r-- 1 user user    14508 2010-09-19 16:03 tagging.odg
```

Listing 2.1: Example of files sorting

Current sample listing is ordered alphabetically by name (by default). The user also can sort file lists by any given property, which in certain cases allows required files to be found more quickly (e.g. if ~1000 files are in one directory or search recursively). Also, the ability to applying multiple filter criteria is available. Combining standard Unix tools such as "ls", "cat"[20], "grep"[21] and others could provide nice search results. But here the availability raises doubts: should all users be strong in Unix utilities? A more intuitive and natural way of sorting data is provided by visual explorers. The user still can sort files using different parameters, but with serious limitations (e.g. it is impossible to temporarily hide some types from the output list). This results in time delays in the case of long file lists. File searching in visual explorers is also limited and usually works much slower than command line utilities.

Another tricky and efficient feature is linking[12]. The word "link" itself assumes some reference or pointer to some data . With the link feature, the user is able to create relatively more complex data organizations. It is an effective and simple way to save some space and synchronize the data, because the user does not have to copy the same files into multiple places. It is enough to store one file in a certain location and make references from the others. Several types of links exist in HFSs, depending on the platform. It is used to divide these types into two major categories: hard and soft links.

A Hard Link[9] is a type of link which points to a file itself. Each target must exist. The traditional Unix style of creating hard links implies setting the same *inode*[1] number to all references. A Hard link itself can also have a different name than its target. So, in this case, no information is provided about the original name of the target. Also, it is not possible to create a hard link for directories (only the root user is allowed with an additional flag for safety) in most cases due to system restrictions (to prevent recursive loops). Another problem with hard links is about sharing attributes of the target, such as size (total directory size with a hard link is increased, as if there would be a normal file inside).

A Symbolic Link[17] (also called soft link) is a type of link which points to a file name. As with the hard links, this type can also have a different name that the target. A soft link contains a relative or absolute path to the target and can also be pointed towards directories. The OS and its utilities can determine the symbolic links and not follow into recursion. A soft link covers most of what a hard link lacks, but in case a target object is renamed/moved/deleted, it stops working because the path string does not change.

Higher level utilities also exist, such as Google Desktop[6], Windows search (comes with Vista or later versions of Windows), Spotlight and others (see some a list of desktop engines on http://en.wikipedia.org/wiki/List_of_search_engines). These desktop utilities allow impressively fast searching to be performed through user space, and additionally provide some custom features. For example, one of Google Desktop's features is a sidebar. It allows one to view e-mail and news, and talk with other users through Google Talk[2] and view RSS[3] feeds. With the Google Desktop Quick Find feature, users do not have to specify the full name of target in the search-box, it is enough to provide only a part of the name. This search engine is based on a continuous file indexing

[1] *inode - data structure in Unix-like file systems that holds meta information about an object (such as a file, directory)*

[2] Chat for some Google services like email, http://www.google.com/talk/

[3] RSS - Really Simple Syndication, http://en.wikipedia.org/wiki/RSS

mechanism and can find the six most relevant results (by default) from the user's local disc. However, high-level utilities like Google Desktop are focused only on searching and the results are shown within the bounds of current utility output. These utilities also do not have a mechanism for the bidirectional relationship between objects, but instead they provide a description of targets (which is only interpretable in the context of the current utility).

One of the most relevant works by description is the Tagstore[19] project. Unfortunately, there is no download link for source or binaries provided[4]. From the description of Tagstore, it follows that it is an open-source project (however, the source as binaries are hidden). The main feature of Tagstore is using so called "tag trees" (hierarchical tags) to navigate the user more effectively through his data tree. Due to missing binaries or source code for testing, it is hard make a judgment about its disadvantages.

2.2 Analysis of opinions

This section presents the discussions and opinions of some people regarding data organization and the sorting problem[11]:

"Rich Kilmer:

What do you store in the namespace to allow applications to cross each others' borders? An agreed-upon ontology is necessary to move beyond today's mess."

In other words, Rich Kilmer raised the problem which is illustrated in listing 2.2. In most cases there are numerous file collections present in the user's space.

```
/home/user
    |-video
    |--films
    |---wild wild west.avi
    |--music
    |---Aerosmith - I Don't Wanna Miss A Thing (Armageddon).avi
    |--birthday.avi
    |--party.avi
    |-music
    |--clips
    |---Aerosmith - I Don't Wanna Miss A Thing (Armageddon).avi
    |--Aerosmith - Crazy.mp3
```

Listing 2.2: Simple types overlapping

Assume we want to add into our collection the file "Aerosmith - I Don't Wanna Miss A Thing (Armageddon).avi". There are two possible locations that exist: "/home-/user/video/music" and " /home/user/music/clips". How to determine which location is actually suitable? It is possible to reorganize our structure by placing, for instance, "/home/user/music" into " /home/user/video/music" since we have two identical folders. However, this solution is not very efficient, for at least two reasons: 1). we have to think about logical reorganization 2). in " /home/user/music" there is another file type (.mp3) which is not related to the video directly. We can also leave the current home-folder's structure "as is" and put into the first location (video) a link, and into

[4]A demo version was requested from Tagstore support at the beginning of April 2011. However, no response was received.

another location (music) the file itself. Corresponding to link type we have different troubles. So, what else can be done? Here is an idea from *Dominic Amann:*

"Start with an efficient file systems that allows small files (such as ReiserFS). Then add an OS browser/shell level extension that allows each folder to contain a special object. This object is a viewer/file systems "plugin" that tells the shell/browser which indexes are available for the folder, and the shell/browser can decide how to display them.

This would allow e-mail to be viewed by a variety of programs, and searchable/useable even by non-email apps because, for example, /var/spool/mail/dominic/ appears to contain

./thread ./subject ./date ./to ./from ./keywords"

Dominic Amann proposed the idea of mixing multiple spaces, which will allow users to work with their different data types across all programs. Each object (it could be even little file structure) has a fixed meta-data about its location, type and special type properties (regarding type). A new file system could significantly increase the efficiency of a user's different interaction activities with information (like simple file searching). For example, a person looks for file F, which is actually located somewhere in e-mails on server M1 (see listing 2.3).

```
united FS
    |-home directory
        |-video
        |-documents
        |-mails
            |-from server M1
            |    |-...
            |-from server M2
                |-from: mail from Ted
                |-to: ...
                |-subject: hi pal!
                |-body: some content
                |-attachments: important file (F)
```

Listing 2.3: Sample file systems view

Now there is no need to perform separate searches through M1 and M2 e-mail servers and local hard disc, since all related data is indexed in the united file systems and is reachable by the user while an Internet connection is up. Once an indexing operation is performed, there is no need to keep an active connection constantly with M1 and M2 mail servers (even an off-line search is available now). These file systems could support some basic simple operation set like CRUD[5]. The only thing that might be needed is data synchronization requests from time to time (deleted or new letters). There is also no need to download files from the email account. Coming back to the previous case in listing 2.2, it is now impossible to create a regular link to file in e-mail space from the local storage and vice versa. Instead of traditional links in file systems, URL can be used.

"Alexander G.M. Smith:

...

[5]CRUD abbreviation coming from words "create", "read", "update" and "delete"

The next step would be to make it (file systems) non-hierarchical. As mentioned elsewhere you want to have relationships bidirectional between a phone number and the person, so a cyclic graph structure of relationships would be needed. Of course, some commands – like "ls -R" – would need to be improved to handle cyclic directories."

Alexander talks here about earlier attempts of user space customization like shown in figure 1.1. The idea Alexander follows is breaking traditional hierarchy of some standard file systems and replacing it with a cyclic graph structure instead, with bidirectional relationships between two objects. It could provide an effective bonus in navigation across multiple locations, so the user can always go back to the start point. Of course, having such a cyclic graph structure assumes certain problems for the standard tools (e.g. mentioned *"ls -R"*). So, there is a need to make these standard utilities behave more flexibly, according to a specific system of current space.

2.3 Introduction to Tagging

A Tag is a form of meta-data[18]. It could be a single keyword (in some cases also a short sentence), or an image, or a specific sound which is assigned to some part of the information. This feature helps to describe an item by referring it to some special set of items with similar properties. The name of a tag expresses the nature of tagged items. Tagging allows searching performance to be increased significantly, thereby reducing the total number of viewed elements (items).

Possible obstacles to the use of tags in searching engines are missing information about the meaning of tags. Relatively similar sets of items could be tagged with different tags. In a listing 2.2 we described a similar problem regarding the relevance of an item. Now, consider the case where instead of one directory "video" there are two similar names: "movies" and "films". Or the case when the user makes a grammatical mistake (or uses singular and plural forms) which leads to the creation of a duplicate tag. Both situations have the same problem: the semantics of all tags are equal and the user can apply both tags. Mentioned obstacles also could lead to possible overloading of tags and the effect of search speedup will disappear, because users have to search through long list of tags beforehand.

Another similar problem with tagging is related to the individuality of the "tag vendor". Of course, the flexibility of tagging allows users to categorize their items in any useful way they can find, but personalized terms can lead to inappropriate relationships between items. This issue can be the reason behind inefficient searches for information about a subject. For example, the tag IT can refer to information technology, or income tax, or Internet television, or the time zone of Iran.

Chapter 3

FUSE

This chapter will continue with related work. It will introduce briefly the definition of virtual file systems along with some examples. Then file systems in user space will be introduced. The introduction part implies:

- platform integrability
- already known virtual file systems that are based on FUSE
- work process description of FUSE
- integration points
- license issues

3.1 Virtual file systems

By definition, a Virtual File System[24] (VFS) is a kind of abstraction layer on top of a more concrete file system. The main purpose of a VFS is to allow client applications to access different types of specific file systems in a uniform way. That means that VFS, for example, can be used to access local and network storage devices transparently, without any difference to the client application. So it does not matter if we keep data in Windows, Mac OS or Unix file systems – the client application can access the data uniformly.

A VFS specifies an interface between the kernel and a specific file system. Therefore, it is easy to add support for new file system types to the kernel simply by implementing the interface. It is possible that VFS can eliminate an incompatibility from release to release. For example, a case where the client application requires a certain version of a specific file system. VFS can even guarantee further stable work with future releases. This means that there are a lot of benefits to using VFS.

Also, VFS sometimes refers to a certain file or even bunch of files that act as a manageable container with the functionality of specific file systems. For example, such containers are SolFS[**?**] or a single-file virtual file systems in an emulator like WinUAE[28], Sun's VirtualBox[25], Microsoft's Virtual PC[27], VMWare[26], etc. The main benefit of this type of VFS is that it is well centralized and easy to remove if need. A single-file VFS can include all the basic features of any specific file

systems, but access to internal structure is often limited. Another drawback of such VFS is low performance because of the high cost of shuffling virtual files when data is written or deleted from virtual file systems.

3.2 About FUSE

The acronym FUSE comes from the words "File systems in User Space". It is a separate executable VFS which was originally developed as AVFS[1], but later became a separate project. It represents itself as a loadable kernel module, basically for Unix-like operating systems. There is also the possibility to apply FUSE in Microsoft Windows, but FUSE does not support the lowest-level file system access application programming interfaces in Windows. Therefore, not all client applications will be able to access file systems that are implemented through FUSE extensions.

Basically, FUSE allows users to create (as an extension) their own customized file systems without kernel code modification. So, actually, FUSE is a "bridge" between user-side created system customization and actual kernel interface.

FUSE main features are:

- Simple library API
- Simple installation (no need to patch or recompile the kernel)
- Secure implementation
- User space - kernel interface is very efficient
- Usable by non-privileged users
- Runs on Linux kernels 2.4.X and 2.6.X
- Has proven very stable over time

Originally FUSE was written in C language, but nowadays quite an impressive variety of other language implementations exist (Java, Python, C#, PHP, Sh, Perl etc). There are many FUSE-based file systems[16] in different categories:

- ArchiveFile systems - accessing files inside archives (tar, cpio, zip, etc.)
- CompressedFile systems - accessing files in a compressed image (gz, zlib, LiveCDs, etc.)
- DatabaseFile systems - storing files in a relational database (MySQL, BerkeleyDB, etc.) or ones allowing searching using tags or SQL queries
- EncryptedFile systems - storing files in a more secure way by using a secret key
- MediaFile systems - storing files on media devices such as cameras and music players or accessing and categorizing media files
- HardwareFile systems - provide access to weird hardware

- MonitoringFile systems - provide notification when a file changes
- NetworkFile systems - storing files on remote computers, including file servers and web sites
- NonNativeFile systems - traditional disk-based file systems that aren't standard on Linux (NTFS, ZFS, etc.)
- UnionFile systems - merging multiple file systems into a single tree
- VersioningFile systems - file systems that remember old versions of files and ones which provide access to version control systems

These are the only know categories (extra FUSE-based projects could be found in addition).

3.3 How does it work?

As it was mentioned in the previous section, FUSE is only a "bridge". The Figure 3.1 illustrates the basic principles of the operation of FUSE. Initially the user writes his custom FUSE extension and runs it with parameters. In the given case we run "example/hello" file within a user specified mounted folder "/tmp/fuse". This means that FUSE will work only inside mounted folder and has no effect on other files outside. There is also the possibility to specify a data folder (by default it takes the same directory where "example/hello" runs). So, basically, the user can control his space with the kernel API through FUSE. The main difficulty is to pick up the required combination of atomic system operations in order to capture more complex actions (some operations, such as copying a file or deleting a folder with content, consist of sequences of other operations).

Sample user file systems in "hello world" style:

```
1  #include <fuse.h>
2  #include ...
3  static const char *hello_str = "Hello World!\n";
4  static const char *hello_path = "/hello";
5  static int hello_getattr(const char *path, struct stat *stbuf)
       {
6  ...if(strcmp(path, hello_path) == 0) {
7  stbuf->st_mode = S_IFREG | 0444;
8  stbuf->st_nlink = 1;
9  stbuf->st_size = strlen(hello_str);
10 }
11 ...
12 }
13 static int hello_readdir(const char *path, void *buf,
       fuse_fill_dir_t filler, off_t offset, struct fuse_file_info
       *fi) {
```

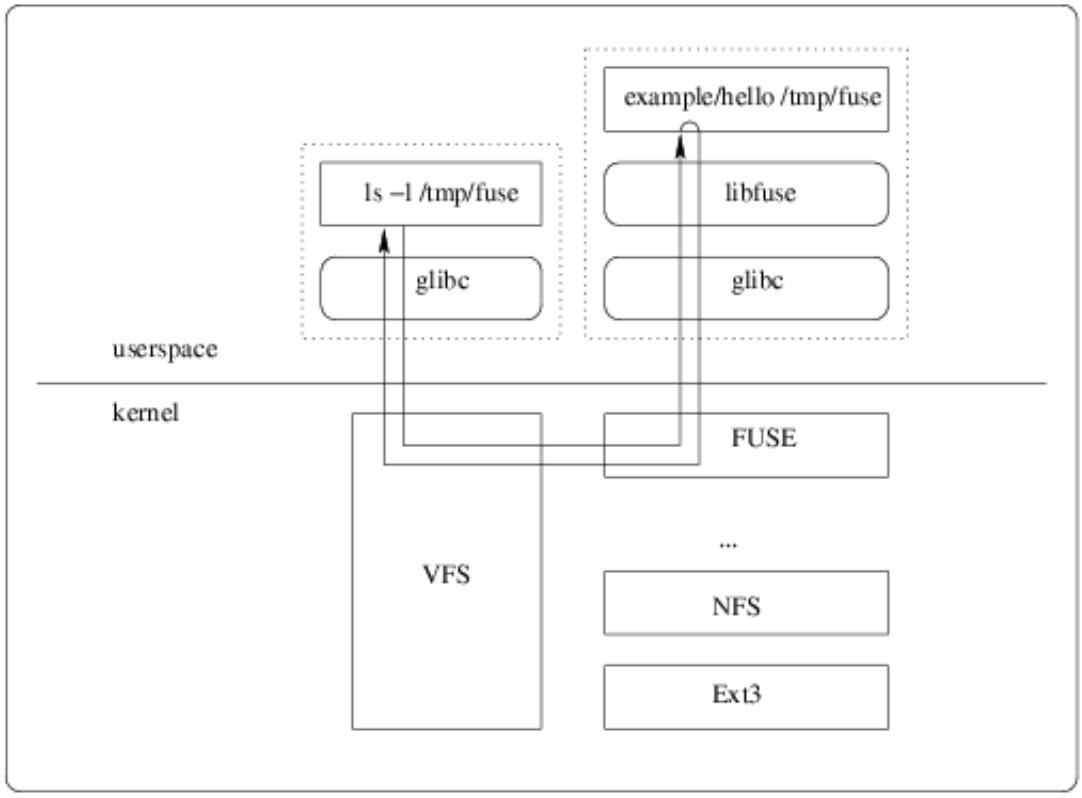


Figure 3.1: FUSE structure (image source [?])

```
14 ...
15 filler(buf, hello_path + 1, NULL, 0);
16 ...
17 }
18 static int hello_open(const char *path, struct fuse_file_info
     *fi) { ... }
19 static int hello_read(const char *path, char *buf, size_t size
     , off_t offset, struct fuse_file_info *fi) {
20 ...
21 if(strcmp(path, hello_path) != 0) return -ENOENT;
22 len = strlen(hello_str);
23 if (offset < len) {
24 if (offset + size > len) size = len - offset;
25 memcpy(buf, hello_str + offset, size);
26 } else size = 0;
27 return size;
28 }
29 ...
```

Listing 3.1: sample FUSE extension

It is quite easy to specify your own customized actions instead of standard file system operations (see FUSE API for full list of operations). In listing 3.1 such "atomic[1]" file

[1] Atomic operations in the FUSE context are those which do not call another operations (e.g. "copy"

```
1 ~/fuse/example$ mkdir /tmp/fuse
2 ~/fuse/example$ ./hello /tmp/fuse
3 ~/fuse/example$ ls -l /tmp/fuse
4 total 0
5 -r--r--r-- 1 root root 13 Jan 1 1970 hello
6 ~/fuse/example$ cat /tmp/fuse/hello Hello World!
7 ~/fuse/example$ fusermount -u /tmp/fuse
8 ~/fuse/example$
```

Listing 3.2: **Example user session**

systems operations, such as getting attributes from a specific object (hello_getattr), reading directory content (hello_readdir), opening (hello_open) and reading (hello_read) file[2] were overridden. The custom prefix could be also specified for methods names. In the current example, the user session[3] will look like Listing 3.2.

After the script "hello" is launched with the starting folder **"/tmp/fuse/"** as a parameter, the user can access the customized user space in the mentioned directory. How to understand how this stuff works? In the custom user space folder, the usual "cat" command actually does not work with a real file, but for input is an output stream from FUSE instead. In a listing 3.1 hello_read custom operation assumes, that if work path = hello_path = "/hello", then hello_str is printed ("Hello World!\n"). This is exactly what we have in listing 3.2for output. So basically, it is always possible to assign any custom content for any path in the user space.

3.4 Licensing issues

The kernel part is released under the GNU GPL[4]. Libfuse is released under the GNU LGPL[5]. All other parts (examples, fusermount, etc) are released under the GNU GPL. This means, that modified versions of code can be sold for money (see http://www.gnu.org/licenses/gpl-faq.html#DoesTheGPLAllowMoney), but the source code should be also provided with binaries (or with first customer request). Also GPL assumes that further modifications of code should be also open-code products and can not be distributed only as binary files. (http://www.gnu.org/licenses/gpl-faq.html#ModifiedJustBinary). Current work uses only Libfuse source. Since Libfuse is under LGPL license, the source code can be hidden from customers and may also be distributed under payment obligation term.

is a complex operation). Generally, FUSE allows only extension of atomic operations.

[2]Note that file operations can be also defined as a separate class.

[3]see sample user session at http://fuse.sourceforge.net/

Chapter 4

Requirements

This chapter introduces the functional and non-functional requirements, gives a brief description of project scope and final product perspective, and provides requirement specification. The last one will be described through use cases.

4.1 Requirements elicitation

The first step to meet the desired outcome for any project or work is to elicit the goals and requirements. First of all, possible stakeholders should be introduced:

- university personal (lecturers, researchers, secretaries, assistants, programmers and others who accumulate data)
- students
- business organizations, companies
- other people (anyone who cares about their data organization efficiency)

There are different techniques existing for requirements elicitation. The stakeholder interview is a commonly used technique, but in the case of current work a specialist's opinion (a person already familiar with the problem) is more likely suitable.

The key ideas of the section 2.2 are captured as the following requirements:

functional requirements

1. File categorization feature (tagging as maintaining speedup)
   (a) bidirectional relationships (for file tagging/categorization)
   (b) Hierarchical categorization of files (category in category)
   (c) Basic file operations support[1] (no regression)
2. Uniform file meta-data representation (for cross-platforms)

[1] FUSE platform assumes the usage of atomic file operations. Any such operation can be customized in a different way. The requirement position tells about not losing functionality for the end-user. For example, the copy operation consists of 1) reading of target location 2) reading of destination directory 3) creation inode structure in destination point 4) filling with relevant content. The requirement assumes that the end-user can copy target files after customization.

(a) index storage of all meta-information

(b) common meta-data structure skeleton per object

(c) Virtual property files support (will be needed for creation of more complex solutions and more file structures will be supported)

nonfunctional requirements

1. Integration: the outcome of current work should be suitable for future cross-platform developments
2. Python implementation
3. FUSE platform based
4. File management should be improved
5. Final product should not slow down an operation's performance below 25%
6. The source code should have explanations or descriptions of functions
7. System should be able to be started/finished within 5 seconds after corresponding command
8. All end-user's operation can be performed at least in the command prompt window.

4.2 Scope

In this work the usability of Unix-like file systems will be improved by FUSE extension.

External deliverables:

1. Generated meta-data for each file (except links)
2. Usability improvement feature: hierarchical tagging

Internal deliverables:

1. Current specification.
2. FUSE extension script in Python programming language
3. Functions descriptions in a source code
4. Additional utilities (like cleaning data from meta-information)

Functionality:

- extension mount/unmount
- meta-data generation
- meta-data access (even unmounted system)
- hierarchical tagging
- traveling opportunity over the tags
- CRUD[2] support as other file systems

Structure: See 3.1 for technical structure.

Assumptions:

1. FUSE will work under Windows platform also (for future modifications)
2. Project will continue
3. Additional generated files will not create a serious trouble for users
4. FUSE is stable
5. Work output is not a final product
6. Most future related work could be relocated to other platform
7. Current project may become a commercial one.

product perspective Current work output is planned as a future base platform for development and research. Also there is a possibility of future commercial outcome (in the long-term perspective, in case of success). Figure 4.1 illustrates the final product perspective. By custom operations and file view assumed Inferato FS integration with user standard file systems operations.

End-user operates withing his customized piece of space, where it is possible having some custom file views, bidirectional files relationships, perform a customized file operations (which nothing more than normal operations adopted to a new container context). A traditional hierarchy could be broken by file categories: file can be located in multiple places in parallel. From the other side, some piece of user space could be out of extension (user may want not to use customization to whole data).

[2]The minimal set of actions with file: create, read, update, delete.

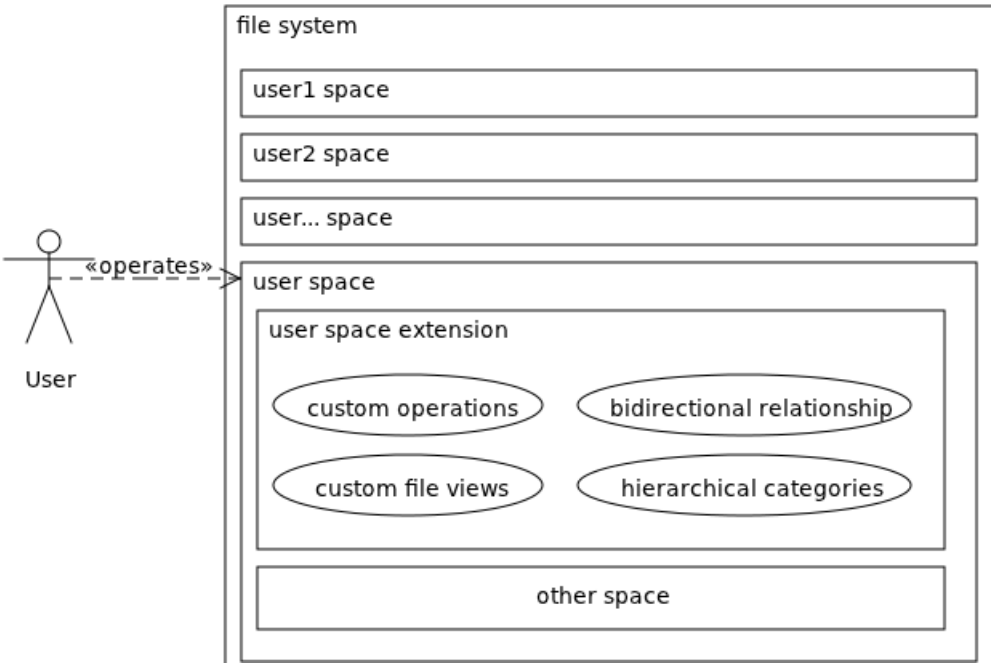


Figure 4.1: Product perspective

4.3 Requirements Specification

The name of delivered file systems is "Inferato FS". It should extend FUSE platform. The list of requirements in section 4.1 implies following use cases:

Id	1
Use case	Object[3] tagging (categorization)
Description	User places tag-folder into meta-folder "tags" of chosen object. Inferato FS tags the file.
Actors	User, extended file systems (Inferato FS)
Dependency	None
Preconditions	In one window user pick the file and opens its meta-folder "tags". In another window user picks the tag-folder. Inferato FS is running.
Postconditions	File is tagged: tag-folder contains a link to tagged file, meta-data of tagged file contains link to tag.
Alternatives of the main scenario	Tagging functionality call could be performed with third party plug-in which is compatible with current Inferato FS.

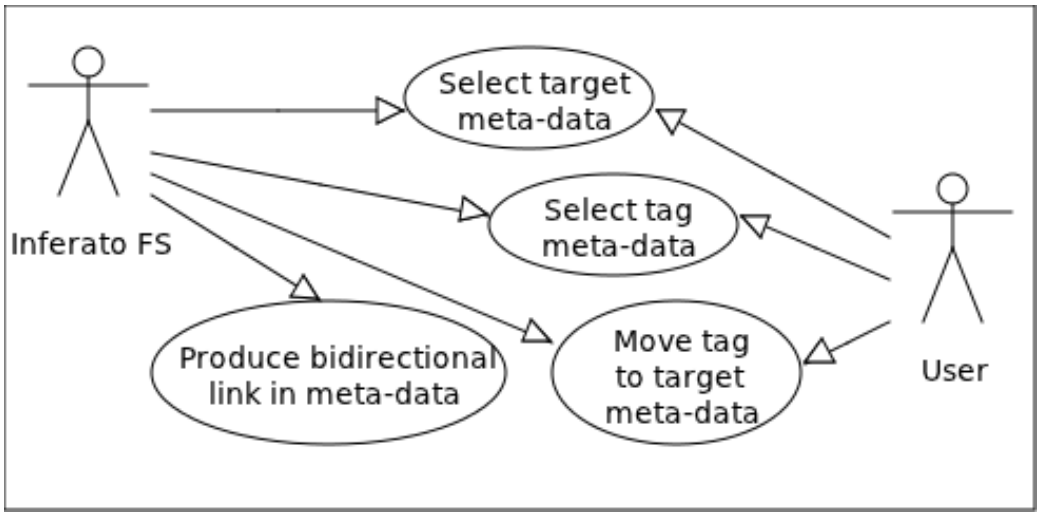


Figure 4.2: Tagging

Id	2
Use case	Object deleting
Description	User deletes the chosen object. Inferato FS validates and delete object with related meta-information
Actors	User, Inferato FS
Dependency	None
Preconditions	User picks an object to delete. Inferato FS is running.
Postconditions	The chosen object is deleted with all meta-information.
Alternatives of the main scenario	There could be any other actor dealing with file deleting instead of user (like client program or some system process).

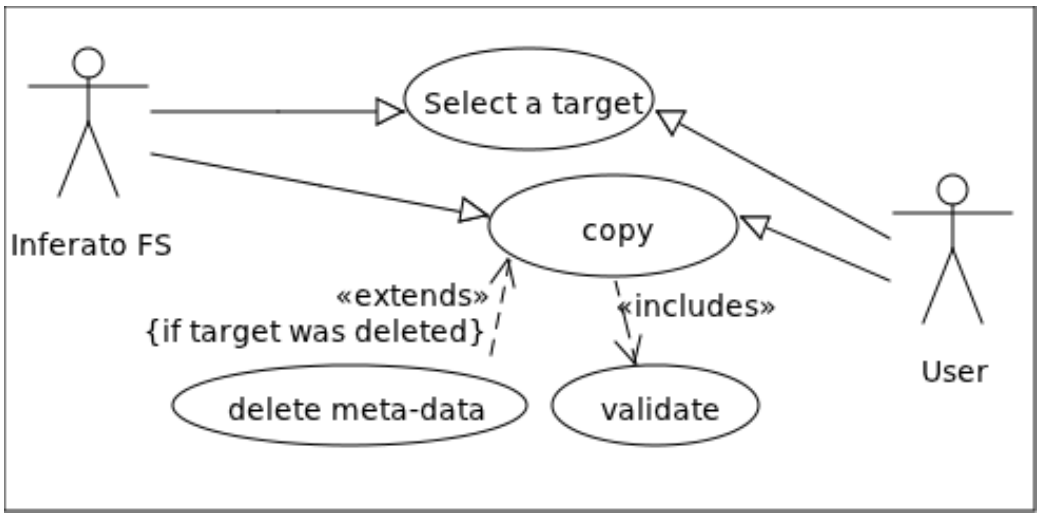


Figure 4.3: Deleting

Id	3
Use case	Object copying
Description	Object duplicate creating.
Actors	User, Inferato FS
Dependency	5
Preconditions	Target object and location for duplicate are chosen. Inferato FS is running.
Postconditions	An object is copied and the corresponding meta-information is created.
Alternatives of the main scenario	There could be any other actor dealing with file copying instead of user (like client program or some system process). User also may do file copy operation while Inferato FS is not running[4].

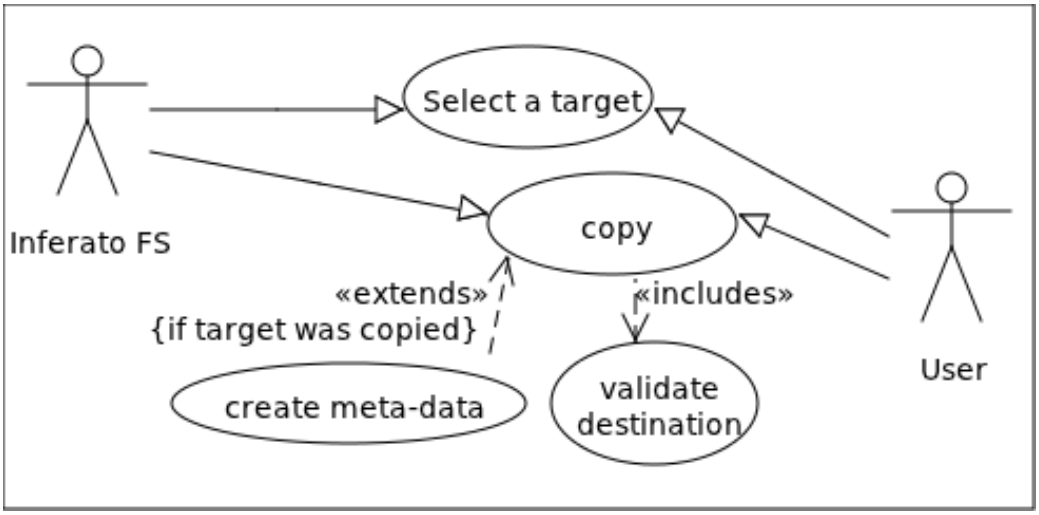


Figure 4.4: Copying

Id	4
Use case	Object moving
Description	User moves object inside his space from one location to another.
Actors	User, Inferato FS
Dependency	2,5
Preconditions	Target object and its new destination directory are chosen. Inferato FS is running.
Postconditions	The path is changed. No file duplications are created. Meta-link stays not affected.
Alternatives of the main scenario	There could be any other actor dealing with file moving instead of user (like client program or some system process). Inferato FS could be not running.

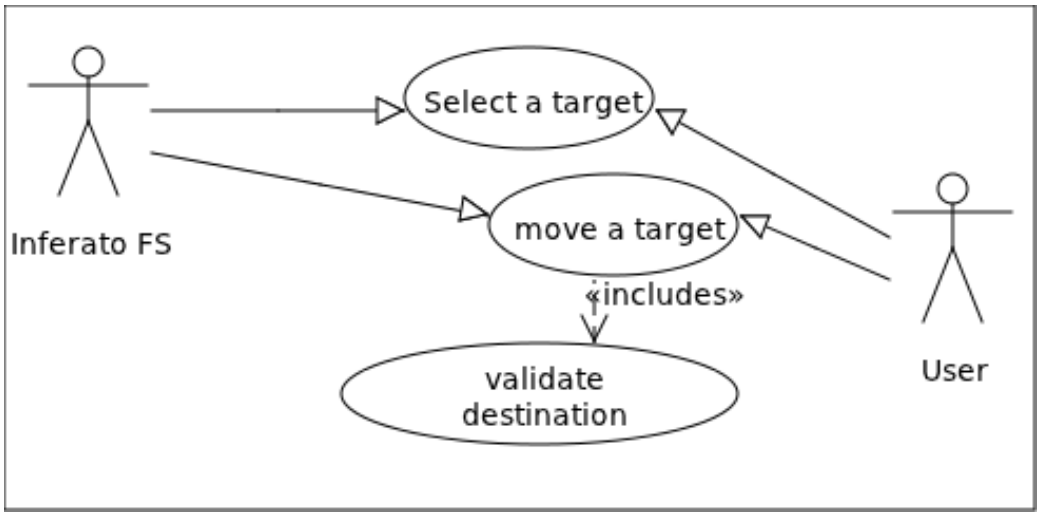


Figure 4.5: Moving

Id	5
Use case	A new object creating
Description	Inferato FS creates corresponding meta-information for newly added/created object.
Actors	User, Inferato FS
Dependency	None
Preconditions	User provides a new object. Inferato FS is running.
Postconditions	The meta-data is created.
Alternatives of the main scenario	There could be any other actor dealing with the object creation instead of user (like client program or some system process). Inferato FS could be not running.

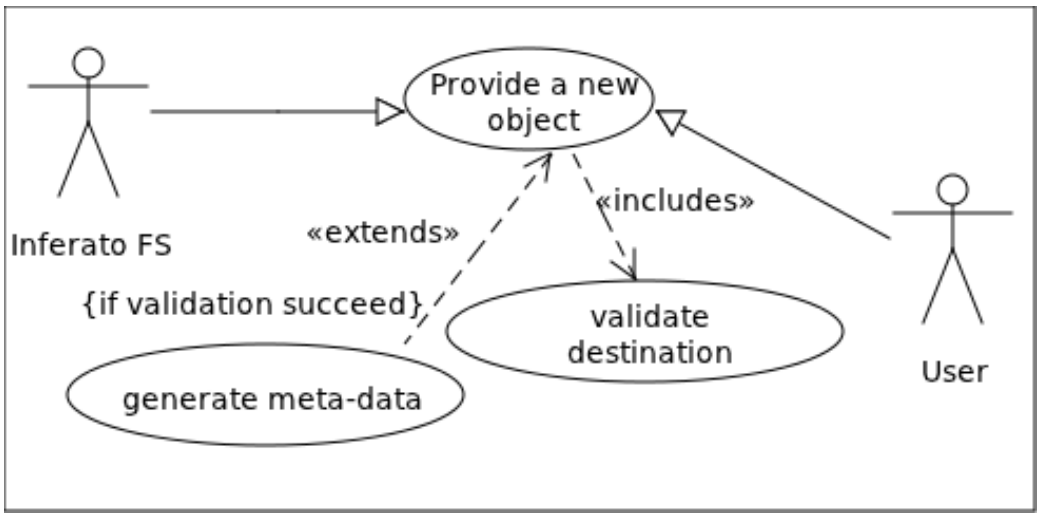


Figure 4.6: Creating

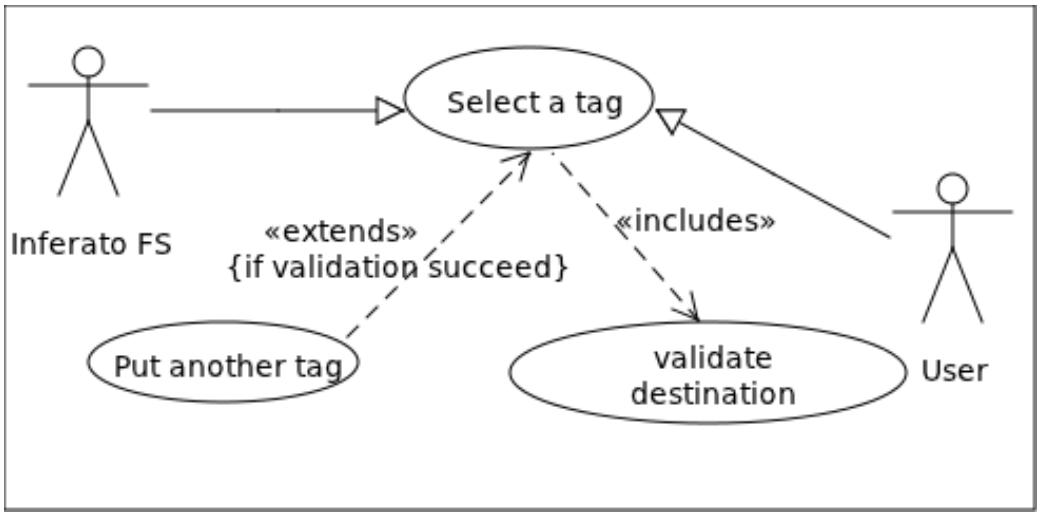


Figure 4.7: Hierarchical tagging

Id	6
Use case	Hierarchical tagging
Description	Each tag can include or be included into another tag.
Actors	User, Inferato FS
Dependency	1
Preconditions	Inferato FS is running. Existing two tags are chosen by user.
Postconditions	Chosen tag appears inside another tag as a new entry.
Alternatives of the main scenario	User can also create a new tag inside the target (which is also a tag).

Chapter 5

Implemented File systems

This chapter describes the implementation of Inferato FS. First of all, integration points with FUSE will be introduced. This is followed by a general description of the implementation of ideas with problematic cases of requirements and proposed solutions. This chapter will also provide a detailed explanation of installation, integration with other tools, and usage process. Followed by a short summary of the advantages and shortcomings of Inferato FS. At the end of the chapter, some tests and future works will be discussed as well.

5.1 Overridden methods

Inferato FS was implemented as an extension to FUSE Python implementation. The name of the base script is "inferatoFS.py". The following methods of FUSE original scripts were overridden:

getattr Getter for attributes of object on provided path (method parameter). Generally, this is the most used method, because other operations are working through it. Current method performs three things:

- ignores some files and folders[1], which FUSE scripts expect to see in the root of mounted system. These expectations are: "autorun.inf", ".Trash", ".Trash-1000", "BDMV", ".xdg-volume-info", ".directory", ".krcdirs", ".kateconfig".
- provides the attributes for regular objects[2].
- in case of Inferato object, method provides customized properties. The objects are: virtual files, meta-links and meta-storage with corresponding content.

readdir This method retrieves the object list from the given path (method parameter) and creates a virtual mirror of entries. Besides regular objects, customization also adds to the mirror image Inferato special files where needed. This method is

[1] On the FUSE homepage some manuals and documentation are provided. However, there is still not enough information provided. The files expected by FS script are not mentioned in documentation pages. Only "autorun.inf" description is provided (last check in April 2011).

[2] Term "regular objects" implies any object in file systems, which is not defined as a part of Inferato FS.

also a key method for future extension based on virtual files[3], since here is defined an initialization of "fakes".

unlink Unlink method serves as a delete operation for objects that are other than the directory. In addition to unlink operations from a standard Python "OS" package, this method also handles an unlinking of the special Inferato meta-structures.

rmdir Originally calls "os.rmdir". As the unlink method, this one is For the deleting operation and called only in case of a directory parameter. The user is not allowed to delete special structures while Inferato FS is running.

symlink A symbolic link creation is handled here. All user links are accepted except those, which can possibly break the normal work of Inferato FS.

rename The tagging functionality is based on this method. Movement of sensitive objects is restricted. Basically, rename operation means not only a target name change, but a path change as well. It is used to refer to path changing as object "moving" and name changing as "renaming", but, in general, these actions are the same. For example, assume that the user has "sample.file" in location " /home/user/". Usage of Unix standard utility "rename" will affect only the end of the path string "/home-/user/sample.file" after last "/", while "mv" can change any part of path (should have corresponding permissions).

chmod For this method, only the sensitive permission changes are restricted. Basically repeats "chmod" utility in file systems.

mkdir Method calls OS system "mkdir"utility. As extension, restricts folder creation in meta-space[4].

access A secure policy can be specified in this method. Currently, extension uses this method calling for the handling of meta-information generation. If an object is not accessed by a user after Inferato FS was started, then no meta-information will be created. The first access is usually performed when reading directory entries (for each entry).

fsinit This method is called only when the system is started. Extension modifications here are related to the initialization of meta-information structures.

Besides customized normal FS operations, abstract file behaviours were also customized. Class InferatoFile implements necessary modifications of FUSE original example. The following method was also overridden:

__init__ The initialization of virtual files are added here.

[3]Virtual files probably will be needed for holding the parameters about the target.

[4]Meta-space - any location or object in Inferato FS, which is used for meta-information

5.2 System design description

Following those requirements described in chapter 4, there is a need to create a type of structure that could maintain the meta-information. On the other hand, it should be available from any location in the mounted system (FUSE mounts a particular folder and acts therein as a separate file system). So Inferato FS uses for a meta-data register its own meta-structure in one single folder named "#[meta-storage]#". Generally, it contains the data structures that describe corresponding files in the user space. In listing 5.1, the example of meta-storage view on a mounted system is provided. After the first mounting, meta-storage will be saved in user data. In the example listing "/home/user/data" path is specified as a root[5] for data and "/home/user/mnt" as a mounting space.

```
/home/user/mnt/#[meta-storage]#
        |-custom tag-f00fcf84-76ad-11e0-b2c2-00215d34df04
        |        |-tags
        |        |-length
        |-folder1-015b0d62-76ae-11e0-b2c2-00215d34df04
        |        |-tags
        |                 |-custom tag
        |        |-length
        ...
```

Listing 5.1: Example of meta-structure

Generally, the structure on example listing contains one unique record per each known file in Inferato FS space. The file is known when and only when the system somehow accesses it. These "lazy" initializations should prevent a long pause in the case of fetching large data. Each record is named by following convention: original object name + unique identifier. The last one is generated with Python "uuid" package by method "uuid1". It basically uses for generation a host ID, sequence number (also seed) and current time. This should be enough to avoid duplicates if Inferato FS migrates to shared systems.

Each record also could contain some properties of the object in the face of virtual files. The example of such property is "length" file. It includes the name length of the target. For example, for "folder1" with 7 characters in name the file "length" will produce "7" for output. Virtual files are only visible in mounted space ("mnt" directory) and their behaviour and output is generated by Inferato FS "on fly". Another entry of meta-record is "tags" folder. Generally, if the target object was somehow tagged, this folder will contain a reference to the tag.

All used names in meta-structures (including virtual files) could be configured through "templates.py" file that comes with Inferato FS distribution.

After the first Inferato FS start, per each object in the same directory there will be generated also a link, which points to the target meta-record location in meta-storage. The name conversion for links is: target name + suffix, where suffix some unique character set for system (initially it is "[#]").

The tagging mechanism implies following key points:

- each folder is a tag

[5]The root of data contains physical meta-data without virtual files, while mount location includes only a fake mirror of data.

- all content inside a directory is automatically tagged by it
- usage of tagging is only for cases when there is a need to break a traditional hierarchy (tree structure)

There is also one special case when the system tries to tag two targets with the same name. This could potentially make the collisions in Inferato FS work. In listing 5.2 is shown a situation, when the user may want to perform tagging of two (or more) identical names.

```
/home/user/mnt/
    |-Downloads
    |    |-Hans Zimmer feat. Lisa Gerrard - Now We Are Free.mp3
    |-music
    |    |-Hans Zimmer feat. Lisa Gerrard - Now We Are Free.mp3
    |-soundtracks
    ...
```

Listing 5.2: Names conflict

Assume that the user tagged a file ".../music/Hans Zimmer feat. Lisa Gerrard — Now We Are Free.mp3" as "soundtracks". In the "soundtracks" Folder, a new reference to the target has now appeared. Now, if the user wants to tag another file in the "Downloads" folder, then there will be a name clash, because possible candidate names for the second target file will be the same as the existing one in "soundtracks". These two files may have different content (e.g. total size, sound quality, length). Thus, this name collision should be resolved considering both targets. In order to prevent such conflicts, a reference in the "soundtracks" folder points to a structure, which basically has two references per each duplicate. One reference points to the source directory of the target and another one to the target itself. Name convention: x, x_sourcedir (where x is a non-negative number starting from 0, which defines the order of tagging registration[6]). In a visa-versa situation, when there are two or more duplicate candidates for a target as a tag, Inferato FS does not allow for the creation of two duplicate tags for one target.

Design assumptions

- User does not use a special Inferato FS suffix for meta-links. This may lead to unstable system work while dealing with tag information [7].
- User does not change the meta-records content directly.
- The key generating mechanism for meta-record never generates the duplicates.
- User acts as a "single thread".
- Inferato FS will be used as a shared service for Windows-like systems in order to avoid FUSE lacks.

[6]Numbers are taken sequentially, but, in the case of object deletion, some numbers may be free and will be assigned to new duplicate candidates starting from the lowest one.

[7]However, a special meta-suffix could be used by the user differently from the suffix name position.

5.3 Installation

Since Inferato FS was written and tested under a Linux-like OS, it is highly recommended to use the same OS type. Inferato FS is a Python script and does not require any additional configuration, except the environment. The following packages should be installed:

- Python 2.6 or later (if not installed)

```
user@ubuntu:~$ sudo apt-get install python2.6
```

- python-dev

```
user@ubuntu:~$ sudo apt-get install python2.6-dev
```

- FUSE libraries

This step implies that one of stable version was downloaded from:
"http://sourceforge.net/projects/fuse/files/fuse-2.X/"

```
user@ubuntu:~$ cd Downloads/
user@ubuntu:~/Downloads$ tar -xf fuse-2.8.5.tar.gz
user@ubuntu:~/Downloads$ cd fuse-2.8.5/
user@ubuntu:~/Downloads/fuse-2.8.5$ ./configure
user@ubuntu:~/Downloads/fuse-2.8.5$ make
```

- Fuse Python

```
user@ubuntu:~$ sudo apt-get install python-fuse
```

After this step, the scrip "inferatoFS.py" will be able to run. Environment installation is now complete. Note that distribution packages for your location may be different from examples.

5.4 Usage

Inferato FS startup. First of all, the user should decide which data system should be used and which folder should be used for a mounted system (should be empty). Assume that Inferato FS installation path is $inf_home = "/home/user/inferato", data folder $data = "/home/user/data" and mounting point $mnt = "/home/user/mnt". Then sample startup session with few debug messages[8] in terminal window will look as follows:

```
user@ubuntu:~$ python $inf_home/inferatoFS.py -f -o root=$data
    $mnt
creating meta-storage '/home/user/data' ...
...done
```

[8]Specifying the flag -f is a kind of trick to reduce the amount of debug information from FUSE.

If you do not want to receive any messages:

```
user@ubuntu:~$ python $inferato_home/inferatoFS.py -o root=
    $data $mnt
```

Starting "inferatoFS.py" with -d flag will allow the complete debug information to be printed. After the system is started, each object in user data will have a generated duplicate with suffix "[#]". This is a link to meta-information of target. As long as this script stays running, in $mnt path will be reflected exact virtual copy of $data path content. A new folder "#[meta-storage]#" contains all related meta-information. Now open another terminal window and go to $mnt path. If everything is done correctly, the session will look as follows:

```
user@ubuntu:~$ ls $data
f1  f1[#]  f2  f2[#]  #[meta-storage]#  tag  tag[#]  target
    target[#]
user@ubuntu:~$ ls $mnt/
f1  f1[#]  f2  f2[#]  #[meta-storage]#  tag  tag[#]  target
    target[#]
user@ubuntu:~$ cd $mnt/
```

Target meta-data request. Assume now that the user wants to get more information about "$mnt/f1" folder. It is possible to see the tags and one property file "length", which tells the actual number of characters in the name of the target. This is done as an example of a virtual file for programmers who will develop Inferato FS in the future. The sample session will look as follows:

```
user@ubuntu:~/mnt$ ls f1\[#\]/ length  tags
user@ubuntu:~/mnt$ ls f1\[#\]/tags/
user@ubuntu:~/mnt$ cat f1\[#\]/ length  tags/
user@ubuntu:~/mnt$ cat f1\[#\]/length
2
```

or

```
1  user@ubuntu:~/mnt$ ls \#\[meta-storage\]#/ -l
2  total 16
3  drwxr-xr-x 3 user user 4096 2011-05-15 19:15 f1-82771e00-7
       f0e-11e0-b690-00215d34df04
4  drwxr-xr-x 3 user user 4096 2011-05-15 19:15 f2-82784a8c-7
       f0e-11e0-b690-00215d34df04
5  drwxr-xr-x 3 user user 4096 2011-05-15 19:15 tag-8277e6b4-7
       f0e-11e0-b690-00215d34df04
6  drwxr-xr-x 3 user user 4096 2011-05-15 19:15 target-827782b4
       -7f0e-11e0-b690-00215d34df04
7  user@ubuntu:~/mnt$ ls \#\[meta-storage\]#/f1-82771e00-7f0e
       -11e0-b690-00215d34df04/
8  length  tags
9  user@ubuntu:~/mnt$ cat \#\[meta-storage\]#/f1-82771e00-7f0e
       -11e0-b690-00215d34df04/length 2
```

Tagging. There are two possible ways of tagging Inferato FS supports. The first way provides the opportunity to tag one target with one or multiple tags, the second one – visa-versa. Assume that the user wants to mark "target" folder as "f1" and "f2". Sample session[9]:

```
1  user@ubuntu:~/mnt$ mv -t tag\[#\]/tags f1 f2
2  user@ubuntu:~/mnt$ ls tag\[#\]/tags/ f1  f2
3  user@ubuntu:~/mnt$ ls tag\[#\]/tags/ -l
4  total 0
5  lrwxrwxrwx 1 user user 30 2011-05-15 23:39 f1 -> /home/user/
       data/f1
6  lrwxrwxrwx 1 user user 30 2011-05-15 23:39 f2 -> /home/user/
       data/f2
7  user@ubuntu:~/mnt$ ls f1 -l
8  total 4
9  drwxr-xr-x 2 user user 4096 2011-05-09 00:15 tag
10 drwxrwxrwx 5 root root 0 1970-01-01 03:00 tag[#]
11 drwxrwxrwx 5 root root 0 1970-01-01 03:00 target[#]
12 user@ubuntu:~/mnt$ ls f2 -l
13 total 4
14 drwxr-xr-x 2 user user 4096 2011-05-08 22:17 tag
15 drwxrwxrwx 5 root root 0 1970-01-01 03:00 tag[#]
16 drwxrwxrwx 5 root root 0 1970-01-01 03:00 target[#]
```

Assume that user did not performed previous step and now wants to tag objects "f1" and "f2" as "target". The second way of tagging will look as follows:

```
user@ubuntu:~/mnt$ mv -t target\[#\]/ f1 f2
user@ubuntu:~/mnt$ ls target
f1[#]  f2[#]
user@ubuntu:~/mnt$ ls f1\[#\]/tags/
target
user@ubuntu:~/mnt$ ls f2\[#\]/tags/
target
```

Untagging. As is the case with tagging, an untag action can be done in two ways. For the untag operation a user needs to delete the meta-link. The following sample session covers both cases:

```
user@ubuntu:~/mnt$ rm f1\[#\]/tags/target
user@ubuntu:~/mnt$ ls f1\[#\]/tags
user@ubuntu:~/mnt$ ls target f2[#]
user@ubuntu:~/mnt$ rm target/f2\[#\]/
rm: cannot remove 'target/f2[#]/': Is a directory
user@ubuntu:~/mnt$ rmdir f2 f2/    f2[#]/
user@ubuntu:~/mnt$ rmdir target/f2\[#\]/
user@ubuntu:~/mnt$ ls target
user@ubuntu:~/mnt$ ls f2\[#\]/tags/
```

[9]In this example both "f1" and "f2" directories have an entry "tag" which is nothing more than a regular folder.

TThe reason why meta-links are shown as directories is as follows: in case two or more targets with the same name were tagged by one tag, then meta-links points to structure, where all duplicates are described. So, basically, it could be compared with a folder, which stores all "incoming" references (see figure 1.1).

Other file operations are available as in a normal file system, except in cases dealing with special data.

Unmount. If a user wants to unmount Inferato FS, then all opened files/directories should be closed. Otherwise the system will print a corresponding message:

```
user@ubuntu:~$ fusermount -u mnt
umount: /home/user/mnt: device is busy.
(In some cases useful info about processes that use
the device is found by lsof(8) or fuser(1))

//closing all work and trying again

user@ubuntu:~$ fusermount -u mnt
user@ubuntu:~$
```

Data cleaning. Distribution archive of Inferato FS includes script "fsclean.py". This allows a user to clean his data:

```
user@ubuntu:~$ python $inf_home/fsclean.py $data
Cleaning is done
user@ubuntu:~$ ls $data
f1  f2  tag  target
```

5.5 Integration with Graph3d

For simple test purposes, Inferato FS was integrated with Graph3d[10]. This is a simple 3d browser, which could shows different structures through graph structure. For example, different data trees in a file system, or social network connections between people, or web-pages and links between them. Originally written on Panda 3d[14] engine by Dmitri Danilov as a part of his master thesis.

In the context of Inferato FS, the integration with Graph 3d means an opportunity to demonstrate how flexible this file system could be for other utilities. Graph3d itself does not support any file operations. It only can navigate through a user's data tree.

5.6 Advantages

- User does not have to go deeply into data hierarchy, but instead, he can create custom views in a root of the file system. Each view can contain references to data from a different hierarchical level.

[10] Grapth3d repository link: svn://www.dougdevel.org/misc/publications/theses/Master/DmitriDanilov

- All tags are always up-to-date. There are no broken links. If a user deletes or moves a target to another location, all meta-data automatically changes.
- Intuitive and natural way while working. Inferato FS mixes an existing hierarchy of data. For example, users do not have to specify that all files inside the directory "/home/user/Video" should be tagged as "Video".
- Data safety. Inferato FS do not interact directly with user data. It only affects meta-data in addition to user actions.
- User always knows which files are tagged by target and visa-versa.
- Meta-content is available as regular data. User can browse through his files via non-integrated file browsers and use the features of Inferato FS.

5.7 Disadvantages

The purpose of the current list of disadvantages is not only to show the possible lacks of concepts. Some disadvantages are actually "to be done" features that this works assumes.

Project scope

- FUSE related risks: current work is based on FUSE platform and this brings some restrictions (see section 3.4 for more details). It may be necessary to overwrite some FUSE part, for example "fusermount". In this case, GNU license restrictions for open source software will apply.
- FUSE is not fully functional in Windows family operating systems (see section 3.2 for more details).
- The complete list of meta-properties for system is unknown yet.

Implementation

- Implementation in Python. From the optimal performance it is more likely to use C++ programming language. In case of big user data structures the final performance could have a sensitive difference.
- Only a prototype. There is a need to do a lot of tests and patches for providing more stable system work before real users start trying it.
- Operation back up mechanism for Inferato FS specific operation is missing, as missing user action tracking utility.
- No suitable user interface yet. All tagging actions are performed manually.
- The maximum length of meta-structure names is actually shorter (for 37 symbols) than usual file name capacity assumes.

5.8 Tests and results

Inferato FS work was tested with different amounts of data. Basic functionality was tested with the small data tree and did not not show serious troubles in work. Also, a real data collection was given as input. Usability test was done with a real collection of files (my university materials, collected from 2003 to 2010). A total input of 35 188 files and 9 557 Directories, with a total size approximately 2.2 Gb. It was a good experience to tag the real files. The major problem in usage was that an effective user interface is not available yet. Despite this shortcoming, the system demonstrated itself as an effective method of organizing data. My experience showed, that conception "each folder is a tag" was working perfectly for me. For test purposes, I created some custom views from different programming language materials and this view has covered many of my university courses.

5.9 Planned future works

User interface. Inferato FS needs a suitable user interface (UI), which will provide a more flexible and faster way to tag/untag files. At the moment, each file has a meta-link which points to meta-data. It would be perfect if these links were hidden from the user. The generation of these links is possible on-the-fly and should not be a big problem for UI implementation.

BACKUP mechanism. Currently, Inferato FS supports a basic actions tracking utility. No restoring feature is implemented yet.

New architecture. With suitable UI there will be a great possibility to physically eliminate the meta-links. Meta-storage could also be placed into each directory. For example, it could be more flexible to store in each directory meta-storage as hidden ".meta-storage". There are three major advantages of such architecture:

1. Such meta-storage splitting will prevent possible problems with performance when a user is trying to access it. For example, this is very actual in case of having 100 000 files.

2. There will be no need to generate meta-links, because each meta-storage will hold only meta-data for a particular folder. Thus, it will be easy for UI to generate a pointer to meta-information of target.

3. The need for a unique name in meta-storage will be obsolete. Usually, hierarchical file systems assume that no duplicate names can exist in the same path. Thus, there will be no duplicates in meta-storage either.

Summary and outlook

The main goal of this work is to prove the possibility of the existence of integrative file systems with bidirectional linking and to show the advantages of such a feature.

The work analysed and evaluated research of similar existing approaches and presented an own solution based on the FUSE (File system in User Space) extension platform. The design of the solution is flexible and supports other add-on modifications to the current system. This will allow, in the future, for the extension of the project to handle more effectively complex data structures in a graph based file system.

The first chapter of this work introduced the thesis and explained motivations behind it with the help of a sample scenario. The second and third chapters were dedicated to related work. The core platform was introduced here. The fourth chapter covered the requirements related issues. The fifth chapter gave detailed explanations about the implemented file system, its advantages and shortcomings, tests and future works.

The main goal of this work was met. It was proved that such file systems with bidirectional linking can exist. The advantages of such feature were presented. The results and output of the current work will be used as a development base for commercial projects.

Integreeritavad failisüsteemid

Magistritöö (30 EAP)

Dmitri Borissenko

Resumee

Käesoleva töö peaeesmärk on tõestada kahesuunalise linkimisega integreeritavate failisüsteemide võimalikkust ning tuua esile sellise funktsiooni eelised.

Töö käigus analüüsiti ning hinnati uuringuid sarnaste olemasolevate lähenemiste kohta ning esitati omapoolne lahendus, mis põhineb FUSE (File System in User Space – kasutajakeskkonnas olev failisüsteem) lisaplatvormil. Lahenduse disain on paindlik ning toetab teisi praeguse süsteemi lisand- modifikatsioone. See võimaldab tulevikus projekti laiendada, et tegeleda tõhusemalt keeruliste andmestruktuuridega graafikal baseeruvas failisüsteemis.

Käesoleva töö esimeses osas selgitatakse töö eesmärki ning selle valiku põhjuseid põhinedes näidisstsenaariumile. Teine ja kolmas osa on pühendatud seotud tööde läbi viimise kirjeldamisele. Siin tutvustatakse ka põhiplatvormi. Nõudmistega seotud küsimuste osas antakse ülevaade neljandas osas. Viiendas osas selgitatakse põhjalikumalt failisüsteemi rakendamist, selle eeliseid ja puuduseid, testimist ja sellega seotud edasist tööd.

Käesoleva töö peaeesmärk saavutati. Tõestati, et selline kahesuunalise linkimisega failisüsteem saab olemas olla. Välja toodi selle funktsiooni eelised. Töö tulemusi ning väljundit kasutatakse tulevikus alusena kommertsprojektide arendamisel.

Abstract

The main concept of the proposed Integrative Graph File Systems is based on bidirectional relationship between two objects (bidirectional linking). The main features are up-to-date links, no broken references, and improved organization of existing file hierarchy. Nowadays, it is hard to maintain the variety of a constantly increasing number of files. Over time, even a simple file can be lost in the deep hierarchy of user files. With the work proposed here, it is possible to prevent such a loss by offering different ways to traverse the hierarchies while still ending up at the same file. This method is similar to tagging. The work allows the user to easily place a single file in multiple locations on meta-info level and quickly find the incoming links. Thus, the user always knows all objects which are pointing to the target and vice-versa. All basic file operations are supported (like delete, move or rename). The main goals of this work are to prove the possibility of the existence of integrative file systems with bidirectional linking and to show the advantages of such a feature. Integration means the possibility to use such a file system with the standard file operation and exploration tools available in current operating systems. The work analyzes and evaluates research of similar approaches and presents an own solution, based on the FUSE (File system in User Space) extension platform. This solution is applied to several example scenarios. The design supports other add-on modifications to the current system, allowing the extension of the project to unify and sort different data in a graph based file system. As this is an integrative approach, no explicit user interface will be provided. The future work will hint at possible extensions to a collaborative multi-user file system, which assumes the combination of local space and different network or cloud based data providers.

Bibliography

[1] AVFS - a virtual file system. http://avf.sourceforge.net/[Last accessed on May 20, 2011].

[2] C++ language tutorial - c++ documentation. http://www.cplusplus.com/doc/tutorial/[Last accessed on May 20, 2011].

[3] Eclipse - the eclipse foundation open source community website. http://www.eclipse.org/[Last accessed on May 20, 2011].

[4] The GNU general public license v3.0 - GNU project - free software foundation (FSF). http://www.gnu.org/licenses/gpl.html[Last accessed on May 20, 2011].

[5] GNU lesser general public license v3.0 - GNU project - free software foundation (FSF). http://www.gnu.org/licenses/lgpl.html[Last accessed on May 20, 2011].

[6] Google. http://www.google.com/[Last accessed on May 20, 2011].

[7] Google desktop - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Google_Desktop[Last accessed on May 20, 2011].

[8] Graph theory - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Graph_theory[Last accessed on May 20, 2011].

[9] Hard link - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Hard_link[Last accessed on May 20, 2011].

[10] Hierarchical file system - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Hierarchical_File_System[Last accessed on May 20, 2011].

[11] Jon udell: The future of the file system. http://jonudell.net/bytecols/2001-05-30.html[Last accessed on May 20, 2011].

[12] Link - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Link[Last accessed on May 20, 2011].

[13] Mashup (web application hybrid) - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Mashup_%28web_application_hybrid%29[Last accessed on May 20, 2011].

[14] Panda3D - free 3D game engine. http://www.panda3d.org/[Last accessed on May 20, 2011].

[15] Python programming language - official website. http://www.python.org/[Last accessed on May 20, 2011].

[16] SourceForge.net: FileSystems - file systems using fuse. http://sourceforge.net/apps/mediawiki/fuse/index.php?title=FileSystems[Last accessed on May 20, 2011].

[17] Symbolic link - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Symbolic_link[Last accessed on May 20, 2011].

[18] Tag (metadata) - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Tag_%28metadata%29[Last accessed on May 20, 2011].

[19] tagstore - a new way of storing and accessing files. http://tagstore.ist.tugraz.at/[Last accessed on May 20, 2011].

[20] UNIX man pages : cat (). http://unixhelp.ed.ac.uk/CGI/man-cgi?cat[Last accessed on May 20, 2011].

[21] UNIX man pages : grep (). http://unixhelp.ed.ac.uk/CGI/man-cgi?grep[Last accessed on May 20, 2011].

[22] UNIX man pages : ls (). http://unixhelp.ed.ac.uk/CGI/man-cgi?ls[Last accessed on May 20, 2011].

[23] The UNIX system, UNIX system. http://www.unix.org/[Last accessed on May 20, 2011].

[24] Virtual file system - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Virtual_file_system[Last accessed on May 20, 2011].

[25] VirtualBox. http://www.virtualbox.org/[Last accessed on May 20, 2011].

[26] VMware virtualization software for desktops, servers & virtual machines for public and private cloud solutions. http://www.vmware.com/[Last accessed on May 20, 2011].

[27] Windows virtual PC: home page. http://www.microsoft.com/windows/virtual-pc/[Last accessed on May 20, 2011].

[28] WINUAE. http://www.winuae.net/[Last accessed on May 20, 2011].

Appendix A

All related materials (such as source code, example pictures, current writing) could be found in public SVN repository:

"svn://www.dougdevel.org/misc/publications/theses/Master/DmitriBorissenko"

Appendix B

A Gpraph3d integration code:

```
1  from ConfigParser import ConfigParser
2  from parser import Parser
3  from templates import FSTemplates
4  from metautils import PathUtils
5  import networkx as nx
6  import os
7
8  class simpleGraph:
9
10     graph = {}
11     root = None
12     data = None
13
14     def __init__(self):
15         #load properties start
16         cfgfile = os.path.split(__file__)[0] + os.sep + "sys.
               properties"
17         parser = ConfigParser()
18         parser.read(cfgfile)
19         self.root = parser.get("init", "root")
20         self.data = parser.get("init", "dat")
21         #load properties end
22         self.myLoad()
23
24     def getRoot(self):
25         return self.root
26
27     def getNameLengthString(self, path):
28         ret = 0
29         #make sure that this is not a special file and the
               special link exists
30         if not Parser(path).isMeta():
31             length = path + FSTemplates._suffix + os.sep +
                   FSTemplates.file_len
32             file = open(length, "r")
33             #we need to read only the first line
34             if file:
35                 ret = file.readline()
```

```
36          return ret
37
38      def explore(self, arg, dir, files):
39          if not Parser(dir).isMeta():
40              for f in files:
41                  if not Parser(dir + f).isMeta():
42                      if not self.graph.has_node(dir):
43                          self.addNode(dir, "root of FS")
44                      meta = f + " is entry of " + dir + "\n"
45                      meta += "length of name: "
46                      meta += self.getNameLengthString(dir + os.
                            sep + f)
47                      self.addNode(dir + os.sep + f, meta)
48                      self.addEdge((dir, dir + os.sep + f + ""),
                            "")
49
50      def loadRelations(self, arg, dir, files):
51          if not Parser(dir).isMeta():
52              for f in files:
53                  if not Parser(dir + os.sep + f).isMeta():
54                      fmeta_tags = dir + os.sep + f +
                            FSTemplates._suffix + os.sep +
                            FSTemplates.folder_tags
55                          for en in os.listdir(fmeta_tags):
56                          tag = PathUtils(fmeta_tags + os.sep +
                                en).getRealPath(self.data, self.
                                root)
57                          self.addEdge((tag, dir + os.sep + f),
                                "tag")
58
59      def myLoad(self):
60          self.graph = nx.MultiDiGraph(name='File System')
61          os.path.walk(self.root, self.explore, "")
62          os.path.walk(self.root, self.loadRelations, "")
63
64      def getNXGraph(self):
65          return self.graph
66
67      #add node to the graph
68      # index: key
69      # xdata: data in text format
70      # pointers: pointers in text format "1,2,3,4,5,rr,dsf"
71      #point
72      def addNode(self, index, xdata):
73          self.graph.add_node(index, data=xdata)
74          #links
75
76      def addEdge(self, index, type):
```

```
77          self.graph.add_edge(index[0], index[1], data=type)
78
79      def getNodeData(self, index):
80          return self.graph.node[index]['data']
81
82      def getNodeLink(self, index):
83          #return self.graph.node[index]['url']
84          return index
85
86      def getEdgeType(self, index):
87          retArray = []
88          try:
89              list = self.graph[index[0]][index[1]].values()
90              for innerDict in list:
91              retArray.append(innerDict.get('data'))
92          except KeyError:
93              print "ERROR: missing edge index:", index
94              return []
95          return retArray
```