Virtual File System on top of Topic Maps

Alexander Zangerl[1] and Robert Barta[2]

[1] Bond University, School of Information Technology
[2] Austrian Research Centers Seibersdorf

az@bond.edu.au
robert.barta@arcs.ac.at

Abstract. The UNIX file system provides a robust framework to abstract away from technical differences between various storage media. This work summarizes experiences to define a virtual file system on top of an existing topic map. It clarifies the involved concepts, details the mapping and reports on first experiences with a proof-of-concept implementation.

1 Introduction

The UNIX file system paradigm has proven to be a flexible - while relatively efficient – abstraction layer between applications and underlying storage media. Accordingly, the operating system offers access not only to data on disk, but also to other media by organizing byte sequences along a tree structure of directories. As an escape hatch, mainly to break up the rigid tree-structure, the UNIX file system provides additionally uni-directional links from one file to another.

In the past, numerous storage technologies have been successfully implemented, differing in disk usage profile, speed, reliability, etc. The mechanism itself has been extended to pseudo file systems, such as for instance `/proc` or `udev` in Linux kernels, which export process or device characteristics in form of a file-based directory hierarchy.

In this sense a file system is always virtual; applications which follow that ontological commitment of files and directories always have to open files, read and/or write their content and close them afterwards, oblivious of the underlying storage technology and its idiosyncratic consistency rules. Notable examples of such file systems include SSHFS [12] which provides access to a remote file system through an SSH tunnel, and GmailFS [9], which makes a Google GMail mail storage account accessible for general-purpose data storage.

Maicher, L.; Garshol, L. M. (eds.): *Subject-centric computing. Fourth International Conference on Topic Maps Research and Applications, TMRA 2008, Leipzig, Germany, October 16-17, 2008, Revised Selected Papers.* (Leipziger Beiträge zur Informatik: XII) - ISBN 978-3-941152-05-2

Also semantic networks have already been brought into this scheme, in particular using RDF [11] [10]. As the RDF model offers only the construct of a triple, such mapping is rather straightforward. The flatness of the model makes it less useful for humans, though. Using Topic Maps as underlying storage technology is motivated by a number of factors:

- Such an abstraction couches Topic Map constructs in terms of well-known first-class objects such as directories and files. This can lower the intellectual investment necessary to adopt Topic Maps technologies into applications.
- Using files and directories as abstraction layer can dramatically simplify the day-to-day handling of manually managed Topic Maps content. Writing text strings into files is significantly easier and faster for a human user than using a dedicated TM application.
- Navigation through a topic map is translated to navigation through the file system, something which typical command line-oriented interpreters are particularily good at. The same applies to simple lookups into the map (such as *what is the Wikipedia page for concept X*) or simple bulk operations (*give me everything you know about concept X*).
- With a transition from Topic Maps to a file system also the access granularity is changed. Whereas a TM application will have to manage a whole map, broken down to various information items, any access mechanism via a file system has to use a much smaller number of basic concepts.
- On a technological level applications become so independent from any Topic Map library and are effectively TM-API agnostic; and they will continue to operate if the file system uses a completely different technology at some later stage.

The TM file system (TMFS) proposed here covers (a) the translation of TM items belonging to a single map onto a single file system that follows the UNIX file model. This is (b) extended to a translation involving a whole hierarchy of maps.

First it is necessary to clarify the concepts concerning the UNIX file system (section 2). Assuming the reader is familiar with Topic Maps concepts [2] the bidirectional translation between a topic map instance and a single file system is presented in section 3. In a second step *map spheres* [4] [5] are reinterpreted as file systems. In section 4 the practicability of the mapping is demonstrated, along a number of use cases which influenced our design.

To test the approach for feasibility a proof-of-concept implementation based on an existing TM framework has been created. In section 5 some interesting

aspects are elaborated on, so is the list of current deficiencies. This is complemented by discussions regarding the design, scalability and future work (sections 6 to 7).

2 UNIX File System

File systems conforming to the UNIX file model [1] provide a number of mechanisms for organizing content. All are strictly hierarchical and are organized into a single tree of directories. Each directory can contain other directories, (regular) files and a number of special file-like objects:

- *Regular Files* are represented in a file system by a node with a unique internal identifier. Additionally files have a name whereby there are only a few limitations concerning the allowed character set. Every file holds content in form of streams. Specialized files contain only alphanumeric (text) content.
- *Directories* are containers of nodes for files or other directories. Two files/directories must differ in name to make them unique within the directory. Also directories have names and they also have a reference to the parent directory they are in.
- *Symbolic Links* are files in their own right. They are a reference of one file to another (with some limitations) and possibly can also point to a non-existing file. File resolution is normally transparent for applications but they can test files for being symlinks.
- *Mounting* is the process of attaching one storage mechanism to one subtree within a directory structure. All subsequent accesses to that directory or areas below the mount point are translated into accesses to the newly mounted device.

A name within a particular directory can reference a single object only. A named object is either a directory, or a file, be that regular or special. This will guide the conceptual translation of file accesses into the virtual file system to the navigation within a topic map.

Given these types of objects the semantics is determined by operations such as `mkdir` (to create a directory), `rmdir` (to remove it), `open`, `read` and `close` to open, read and close files, and so forth. There are about 2 dozens of these operations, although most of the common functionality is concentrated in a handful of them.

3 Topic Maps to File System Mapping

The design choices for translating a topic map structure into a tree structure were mostly driven by user convenience, not so much by functional completeness. Particular care was taken that the mapping also considers the copious use of text-based notations (such as CTM [8]). This mix opens the pathway to reuse the large existing tool-base for text processing under UNIX (such as grep, find, etc.).

The mapping between the file system tree and Topic Map content is done on two levels. Whole subtrees in a file system can be directly translated into a *map sphere* [5], i.e. a structure which generalizes a single topic map into one which can contain submaps. Individual maps are mapped into one directory.

3.1 Map Spheres

Map spheres are a generalization of a standard Topic Maps structure. Their purpose is to host not only TM content, but also topics of a predefined type which reify further topic maps. As these topic maps can in turn contain further submaps the overall structure takes the form of a tree. To provide a convenient addressing scheme, we adopt a tree-notation of the form `/markup/xml/xslt/`. According to the definition of a map sphere the above expression would locate the top-level map first and would find a topic `markup` in there. That topic is supposed to reify a whole map, also stored inside the map sphere. In that map the topic with the identifier `xml` is expected. With it the same procedure will repeat, as will with `xslt`. At the end of this process a whole map is addressed.

As the processes of *attaching one map* into another is quite similar to *mounting* of file systems, the choice of notation was already inspired by file system navigation.

3.2 Maps

One individual map is represented in the virtual file system as directory. Individual topics will be direct subdirectories thereof, whereby we use topic item identifiers as names for the directory.

For associations we choose a different route in that there exists one directory `.associations` containing all of them. Additionally to associations two more directories are provided: One called `.characteristics`, which contains all names and occurrences, and a second, `.assertions` which contains all of the

above. In all cases an internal association identifier is used as name for a directory.

Apart of these subdirectories the map directory also contains additional *dot* files, such as `.ctm` or `.xtm`. When accessed, they return the map in the respective serialization format. The naming choice with a leading . is deliberate: for many UNIX tools it is a convention to treat such files as *hidden*.

3.3 Topics

Internal topic identifiers are quite convenient to be used as basis for naming the directories for each topic in the file system. They are unique within the underlying map and usually quite short.

Within that directory we bundle all the content pertinent to that topic. For bulk update or retrieval the whole topic information is made accessible as `.xtm` or `.ctm` text files, or any other supported TM serialization format. Again we use a leading dot to indicate to UNIX tools to normally disregard this file. All subject identifiers of a topic are listed in a file named ~, borrowing from the TMQL [6] notation. Similarily any subject locators are stored in a file =. These files always exist, even when there are no locators or identifiers: it is easier to test for empty files than for their existence.

As several names and occurrences can be attached to a single topic, the design choice is to create one directory for each of these, one named `name`, the other `occurrence`; the latter also exists as oc only to reduce typing. Having all names in one directory makes it a trivial operation to find all names.

Each individual name or occurrence is modelled as file, containing the text representation of the value (cf. Fig 1). The name of the file is the topic identifier of the type, `name` and `occurrence` being just two special cases. If either of the names or occurrences are scoped, then the scope (only one single topic as scope is allowed in this scheme) is appended to the file name with `@`, or alternatively using a dot.

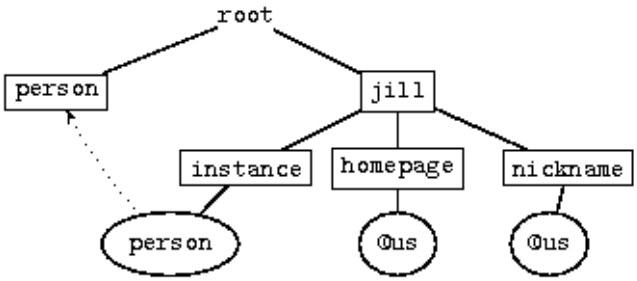

Fig. 1. TMFS structure of a topic

For topics being instances of others, the directory `type` holds symlinks named after the item identifiers of the class topics. For topics used as types, the mechanism is analogous but uses the directory `instance`. For a topic being neither class nor instance, both these directories are empty. Nevertheless these are exposed to an application, because it simplifies later any modification of a topic: a class-instance association can thus be created simply by making a suitable symbolic link in either directory, without having to worry about the directory's existence. As for type-instance configurations between topics, two more directories give access to `superclass` and `subclass` topics.

Regardless whether the topic reifies an association (or a name or occurrence) a file `.reifies` will also exist in the topic directory. If the topic actually does reify any of the above, then `.reifies` will be a symlink to the assertion in question, otherwise the file will be just empty.

3.4 Associations, Names and Occurrences

Associations (but also names and occurrences for that matter) have been tucked away in separate directories, such as `.associations`. In there, each association is itself represented as directory (cf. Fig 2).

In that directory there is always a `.type` symlink to the type of the association, in the same way as there is always a .scope symlink to the scope of the association, defaulting to a topic *universal scope* (us), if necessary.

As associations can be potentially be reified by a topic, the file system procures a `.reifier` file. Also that always exist, being empty if there is no reification, or being a symlink to the topic in question if there is.

The roles quite naturally are organized into a subdirectory each, with the role topic identifier being used as directory name. Within the role directory are then symlinks, one for each player topic pointing to that topic. Here also lies the one difference between fully-fledged associations and names or occurrences: the latter two also have a file `.equates` which symlinks to the file which represents this characteristic under the appropriate topic.

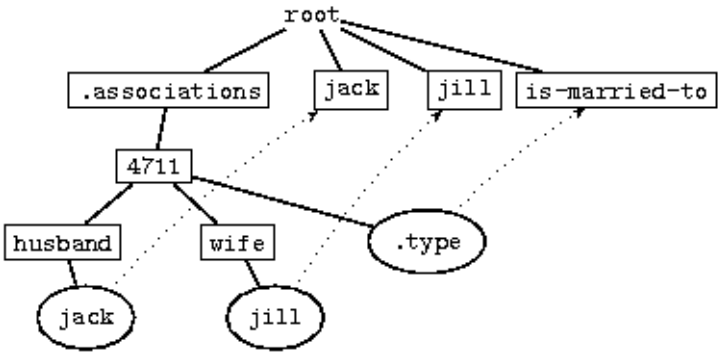

Fig. 2. TMFS structure of an association

4 Canonical Use Cases

In this section we give an overview how Topic Maps, the file system, typical UNIX tools and the compact Topic Map notation together can interact as toolkit to manage Topic Map content at a very small granularity.

4.1 Mounting

In a first step an empty directory has to be created where the topic map structure will reside:

```
mkdir /home/jack/knowledge
```

That directory will act as *mount point* for the topic map content. Everything below that point will be effectively stored in a map. The whole subtree can be provided by a map sphere which has been generated before and is stored in some file, say `/tmp/mapsphere`:

```
mount.tmfs file:/tmp/mapsphere /home/jack/knowledge
```

There are several options concerning the operational model. In some cases the underlying topic map should be used exclusively by the file system. In other cases applications may want to modify the map directly, with changes becoming visible in the file system. Other options control whether map is readonly or not. After mounting the TMFS driver will divert application requests into the file system to the underlying topic map.

To detach the map from the file system, it is *unmounted*:

```
umount.tmfs /home/jack/knowledge
```

4.2 Inspection

After a successful mount a user can list all topics inside the top-level map:

```
cd /home/jack/knowledge/
ls
```

The result list contains directory file names, one for each topic in the map. To find one particular topic based on its internal identifier can be burdened to pattern matching with a regular expressions:

```
ls j*
```

To learn about all names of one particular topic, the user has to find and concatenate all files in the name subdirectory:

```
cat jill/name/*
```

Each name is represented by a single file named after the names' scope, always prefixed by `@`. If the scope happens to be unconstrained, then `us` (the unconstrained scope) is used. Each file contains exactly those names in a particular scope. If `jill` had a `nickname` as type for the name, then it can be retrieved via

```
cat jill/nickname/*
```

or - when the scope is to be fixed -

```
cat jill/nickname/@us
```

In an analogous way the user can learn about all occurrences via

```
ls jill/occurrence/
```

and can extract all, say, of type `homepage`

```
cat jill/homepage/*
```

UNIX tools like find or grep can be used to inspect and query the map. To find all topics which have the `nickname` *chilly jill* grep can used as follows:

```
grep -i "chilly jill" */nickname/*
```

A full-text search for an *interesting concept* over all occurrences can be achieved with some filtering

```
fgrep -l "interesting concept" */occurrence/*
```

To find which topics use the scope `english` for names or occurrences, `find` can be used:

```
find . -name '@english'
```

Also taxonometric information, such as which superclasses, subclasses, types and instances the individual topic has, have been made available as directories:

```
ls joe/type/
ls joe/instance/
ls joe/subclass/
ls joe/superclass/
```

All will return a list of symbolic links leading to all (direct or indirect) types of `joe`, or instances, sub- or superclasses if there were any. If the user prefers to receive the list of types, instance, etc. in one file, then corresponding files exist in the topic directory as well:

```
cat joe/.types
cat joe/.instances
cat joe/.subclasses
cat joe/.superclasses
```

Should a topic have subject locators or identifiers, these can be retrieved via two specially named files. Both contain potentially a list of URIs, or can be empty:

```
cat joe/~
cat joe/=
```

4.3 Bulk Retrieval

In order to simplify bulk extracts from a map, the file system also procures special files which serialize topic map items (topics, names, occurrences and

associations) in one of the serialization syntaxes. To copy all topic information in the underlying map into one CTM document one can write:

```
cat */.ctm > only-topics.ctm
```

To create an XTM document on the fly the user has only to wrap the XTM fragments:

```
(echo "<topicMap>" ; cat */.xtm ; echo "</topicMap>") > only-
topics.xtm
```

Also names and occurrences can be serialized. In

```
cat jack/nickname/.ctm
```

all names of this type, regardless of the scope are collected from this file. To find all names of `jack` and to represent them in AsTMa= (file extension `.atm`), the following suffices

```
cat jack/name/.atm
```

In a similar vein, individual associations offer such files in their respective directory. Also the whole map can be serialized. To do this with the top-level map an application has to read a file in the maps directory, such as

```
cat /home/joe/knowledge/.xtm
```

4.4 Modification

To create a topic in the map, an appropriate subdirectory has to be created:

```
mkdir tmfs
```

First that topic is empty. To add a name in the unconstrained scope, the name string can be directed into the appropriate file:

```
echo "Topic Map File System" > tmfs/name@us
```

Easier it is to use one of the available shorthand text notations and do a bulk insert:

```
cat <<EOT > tmfs/.atm
tmfs isa virtual-file-system-technology
@acronym ! TMFS
! Topic Map File System
EOT
```

The fact that the notation forces to name the topic itself can be used to convey more contextual information about the topic, including associations. This side-effect is quite fortunate as it is not possible to create association directories. If that were allowed, associations without type, scope and roles could be created, something which would lead to an invalid underlying topic map instance.

To guarantee the integrity of associations they are created as *whole*:

```
echo "implemented-with (concept: tmfs, toolkit: perl_tm)"
>> .associations/.ctm
```

It is also possible to incrementally modify associations as long as they always have at least one type and one defined scope, even if it is the unconstrained one.

5 Research Prototype

To test the feasibility and usability of our conceptual translation we have implemented a prototype. It is based on an existing Topic Map distribution, Perl TM [3], and FUSE [13], the *File system in Userspace*.

FUSE is an interface layer between a userland application and the UNIX kernel. It allows to create a file system implementation without having to modify any kernel code. Because of this separation between kernel and the file system code, FUSE has become useful for rapid prototyping and for creating *virtual file systems*, for which there exist a significant number.

Under FUSE the file system handling code is run in user space, with the privileges (and restrictions) of a normal user. Whenever the user accesses the virtual file system, the kernel forwards these accesses to the FUSE process. That is usually encapsulated as a user-executable program which contains the custom callback functions linked to the FUSE library.

On startup this program performs any necessary initialization and mount operations and subsequently enters the FUSE main loop. This FUSE loop listens for notifications of file system accesses from the kernel and schedules the appropriate callback function to implement the particular access.

To produce a particular file system in the FUSE framework one has to provide these callback functions that provide the most common file system-related system calls: `getattr`, `getdir`, `read` and `readlink` are required for read-only access. For write support, the extra functions `mkdir`, `rmdir`, `unlink`, `symlink`, `open` and `write` are necessary. All these callbacks must return the same result types as the respective POSIX system calls of the same name.

As long as the process is active, the file system is available at the specified mount point. The external program fusermount, provided with the FUSE library, is used to finally unmount a FUSE file system; at that point the main loop of the program terminates followed by any necessary cleanup operations.

6 Design Issues

The whole design process and the refinement based on implementation experiences was dominated how to balance out topic map information onto a file system. Obviously structural units, such as individual associations or topics lend themselves to be represented as directories. If you factor in, though, a concise TM notation such as CTM, then much of the Topic Map content can be carried inside text files as well. Crucial in controlling the granularity is whether UNIX tools can more easily access and analyse text fragments, compared to navigating through the directory structure or testing for the existence of certain files.

6.1 Symlinks

There has been some reluctance to readily make use of symlinks to reflect the graph nature of a topic map instance. The problem with symlinks is that they increase the risk of loops inside applications when they navigate through the file system structure. Applications have to be aware of symlinks, so they either ignore them altogether or deploy some loop detection algorithm. The former is something which POSIX tools do by default, but this may also mean that a user will not get the expected behavior.

Another problem with symlinks is that there are several different paths to one and the same topic map item. This is also something the application must be aware of, for instance, when string-comparing topic identifiers. On the other hand, using symbolic links is incredibly useful as it allows quick access to a referenced item.

6.2 Empty Files vs. No Files

When a topic reifies an assertion, then a `.reifies` symlink leads to that very assertion. But if it does not, there is also a `.reifies` file, albeit an empty one.

The reason for this choice is that programmatically testing a directory for emptiness is not a common operation for the usual command-line tools, whereas testing files for emptiness is.

6.3 Occurrences and Names

For both the decision was to organize them according to the type. It was regarded the most discriminant criterion and the chosen form also allows to honor subclassing so that a `nickname` also appears under the `name` directory.

One consequence of offering a directory for every available occurrence and name type is that the files inside these directories can carry the values, and that each of these files have to be named. For names and occurrences with a scope this is obviously the scope name, but for those which live in the *unconstrained scope* a default name (`us`) has to be chosen arbitrarily.

One downside of this approach is that there is no direct way to navigate to the type, or the scope topic. For that, the file name of the type has to be extracted first.

6.4 Associations

Associations (and all assertions for that matter) can be reached via their internal identifier, but that is unlikely to be a very frequent access path. Instead it is expected that applications will use association types and there the symlinks of the `instance` directory to find all associations of that type. A similar approach could have been chosen for scopes, but this was left for a more evolved version of the mapping.

The path can be followed in reverse, although in an unsymmetric way. Every association directory holds a symlink to its type (and its scope).

One notable observation is that associations are not easily de/constructable in an incremental way. There has to be a minimal structure consisting of the association directory, the type and the scope and at least one role before further roles can be added. Otherwise the association would be incomplete, something which cannot be normally represented with an underlying TM infrastructure.

6.5 Taxonomies

One frequent usage pattern is to retrieve the type(s) or instance(s) of a certain topic. While this could be implicitely done via following the appropriate associations, this functionality is so inherent in Topic Maps that it is exposed quite prominently in the mapping, and actually twice to support different access paths.

The first one treats a type (respectively instance, subclass and superclass) as a single entry living in a symlink underneath the `type` (respectively `instance`, etc.) directory. Alternatively the files `.types` (respectively `.instances`, etc.) render the identifiers of all types of that topic. Notable here is that subclass transitivity is always honored. One consequence of this approach is that there cannot be an occurrence or name type called `type` (or `instance`, etc.).

6.6 Feature Completeness

Relative to TMDM [7] the mapping covers all features, with the exception of variants (they can be better modelled with typed names) and the reification of roles (which is arguably a seldom used feature).

6.7 Scalability

The general approach of representing a topic map as relatively flat virtual file system inherently carries some potential pitfalls. One of them has simply to do with the number of topics or associations which translates into a high number of directories within the map or the `.associations` directory.

This presents a twofold problem: first these long lists have to be built on the fly, something which not only stresses the backend storage, but which also asks for significant amount of content to be forwarded between several layers for this one request. This is wasteful, especially if only a fragment of the topics are eventually relevant. Secondly, command-line tools, respectively the UNIX shell, are confronted with these long lists. Here inherent length limits cut off these lists, rendering the whole mechanism useless.

A distinct disadvantage is the circuitous procedure for executing system calls via multiple interfaces. First there is a call path into the kernel to reach the FUSE module. That hands over the request back into userspace to invoke the custom-made semantics. Only then the actual backend - in our case the TM engine will be queried and traversed, if necessary after consulting the persistent backend.

The results are passed back the whole invocation chain. It will have to be seen whether the benefits of TMFS can outweigh the overhead.

7 Future Work

One deficit of the proposed mapping is the lack to address topics via subject identifiers or addresses. URIs are a foreign concept to file systems.

While our implementation experiences using a Perl-based infrastructure were generally positive, our prototype still has several shortcomings on its own:

- Currently it only allows read-access to the underlying map and offers no control over operational modalities. Write-support will further stress-test the usability of the chosen mapping.
- It also does not completely expose the functionality of hierarchically organized topic maps as defined by the map sphere.
- Support for serialization formats such as AsTMa=, XTM 2.0 and CTM are only rudimentary yet.

Apart from building some smaller applications on top of TMFS we also consider to extend the TMFS functionality in several directions. One direction is to find a way to directly add documents themselves to the file system, not only their references into a virtualized topic map. A command

```
cp tmfs-presentation.pdf /home/joe/knowledge/tmfs-
presentation
```

could trigger TMFS into a behavior to not only create the topic `tmfs-presentation` but also to store the PDF document in some background persistent store. The URL generated there for the new document will be used as subject address for the topic.

While useful by itself, it also opens the venue to overlay topic map content with conventional file content. This can be achieved with a *stacked file system* which allows to put one file system onto another. In our case we would start with a conventional file system to store our documents and would overlay a TMFS to provide meta data for these documents.

References

1. *Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities (Volume 1)"*. Information technology—Portable Operating System Interface (POSIX)". pub-IEEE, 1993.
2. International Organization for Standardization, ISO/IEC 13250, Information technology - SGML applications - Topic Maps, 2000.
3. R. Barta. Topic Map Modules for Perl. `http://search.cpan.org/dist/TM/`.
4. R. Barta. TMIP, a RESTful Topic Maps interaction protocol, 2005. Extreme 2005 Montreal.
5. R. Barta. Knowledge-oriented middleware using Topic Maps. 2007. TMRA 2007, Lecture Notes in Computer Science (LNAI) Vol. 4999, Springer 2008.
6. L. M. Garshol and R. Barta. ISO 18048: TMQL, Topic Maps Query Language, committee draft. `http://kill.devc.at/system/files/tmql.pdf`.
7. L. M. Garshol and G. Moore. ISO 13250-2: Topic Maps - data model, 2008-06-03. `http://www.isotopicmaps.org/sam/sam-model/`.
8. G. Hopmans and L. Heuer. Wd 13250-6: Information technology - topic maps - compact syntax, CTM. 2008. `http://www.isotopicmaps.org/ctm/ctm.html`.
9. R. Jones. GmailFS, GMail virtual file system. `http://richard.jones.name/google-hacks/gmailfilesystem/gmail-filesystem-implementation.html`.
10. B. Schandl. SemDAV: A file exchange protocol for the semantic desktop. *Proceedings of the 2nd Semantic Desktop and Social Semantic Collaboration Workshop at the ISWC 2006, Athens, USA, 2006.* `http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-202/SEMDESK2006 0009.pdf`.
11. M. Sintek and G. A. Grimnes. RDF2FS - a Unix file system RDF store. *www.semanticscripting.org, SFSW 2008, 2008.* `http://www.semanticscripting.org/SFSW2008/papers/5.pdf`.
12. M. Szeredi. SSHFS, secure shell filesystem. `http://fuse.sourceforge.net/sshfs.html`.
13. M. Szeredi. Filesystem in USEr space, 2003. `http://fuse.sf.net/`.