HybFS Reference Document

November 12, 2008

Contents

1 Project overview

HybFS is an *almost* semantic file system on Linux that can be overlaid on other file systems and provide multiple organizational views of the same original hierarchy of files, without altering the interface for the existing applications. This is realized by associating multiple *tags* with files. A user can create and assign many tags to a file or groups of files containing other tags. The browsing and file operations can be done by using the concept of *virtual directory*. We can think of a virtual directory as a combination of tags, using logic operators like disjunction, conjunction or negation. This will also be known as a *query*.

In a semantic file system, the navigation is based on the additional metadata associated with files, therefore offering multiple views of the same files. In a hierarchical file system, there is only one organizational view of the data, and a file can be identified in a unique way using this view. We mentioned that HybFS is an *almost* semantic file system, because it actually represents a compromise between the hierarchic and semantic file systems. The user can organize files in a hierarchic way, but in the same time can assign tags, or keywords, to the files. This file system comes with two view perspectives to the user:

- Hierarchic view of the files
- Semantic view of the files

Primary aims

For having a basic functionality, we define the following set of aims that must be completed during the allocated time:

- Implement operations on simple tags and tags with values. A user can create, modify or delete the tags associated to a file.
- Implement support for queries based on multiple tags.
- Mount the semantic file system on top of multiple directories (support multiple branches), regardless of the underlying file system(s), thus offering a unified view based on file metadata.
- Allow operations on sets of files, identified by combinations of tags, also known as queries (for example: delete or move all files corresponding to a query, from the underlying file systems).

Secondary aims

Having considered the limited time, we established a number of features with a lower priority. Still, these are desirable to have, in order to show the potential of a semantic file system.

- Add semantic transducers to automatically extract metadata from new files.
- Add metadata extraction/management policies for each mounted directory/branch, so that new files will be automatically labeled with the set of extracted keywords. As a proof-of-concept, this will be done for MP3 and EXIF metadata.
- Create a separate application for a user-friendly policy management.
- Create a set of scripts to automatically index files by adding the corresponding tags to the file system metadata - used for testing.

2 Design

2.1 HybFS meta-information

Each file can be described by using one or multiple tags. A tag represents an additional attribute set by the user. The tags can be simple, or have an associated value.

Simple tags are specific keywords added to describe a file. The **attribute-value** tags are used to granulate even more the description. In this case, the attribute represents the criteria of description and the value is the subcategory. Ex: the file photo1.jpg can have as simple tags *holiday* and *ski*, and as attribute-value tags *year:2005* , *author:stefania* and *type:jpeg*.

To find the file from the above example, we can specify a path that can be a combination of *holiday* or/and *ski*. To further differentiate between the results of this query, we can also mention the value *2005* for the tag *year* .

In order for the user to have also a hierarchic view, all files in the HybFS have 2 special attribute-value tags:

- `path:file_path (file path without the filename)`
- `filename:file_name (the exact file name as in the hierarchic file system)`

2.2 HybFS semantics

The navigation in HybFS resembles the navigation through a normal hierarchic file system, except the path is in fact a query on the tag information associated with the files. From the application point of view, the file system operations remain unchanged when using HybFS.

2.2.1 Query syntax

The queries will support the following logic operators:

- + - logic AND
- — - logic OR
- ! - logic NOT

Beside logic operators, brackets can be used for more complicated queries.

Ex: ls '(type:photo + year:2005) + (!ski + !snowboard)' - will display all files that have the tag *type:photo* and *year:2005*, but they don't have *ski* and *snowboard* tags.

2.2.2 HybFS operations

Our objective here is to keep the same interface as for the hierarchic file system operations and in the same time to seamlessly add semantic operations.

Concerning navigation, the system will be *2-Dimensional*:

- the current path from the hierarchical file system(s) or:
- the current virtual folder, which retains the current query

HybFS does not permit file names with '(' and ')', due to complex queries that need brackets. If the path is between brackets, it will be interpreted as a query, otherwise it represents a simple file path relative to the hierarchic file system.

2.2.3 File listing

All files will be listed based on a query (if a query exists). If there are no semantic queries, the behavior will be the same as in a hierarchic file system.

One of the possible drawbacks of full semantic file systems is that the user doesn't receive any suggestions when navigating through the files and it has to know what is looking for. This was solved by adding virtual folders to refine the results of the navigation. In our example, when we look for files that are *photos* and have been taken in *2005*, we may not know that these files have other tags also and can be filtered even more. Further more, by having this extra information we can discover other files that share the same information/tags as the listed ones.

Example: Suppose that we want to list all files that have the *type:photo* and *year:2005* tags:

```
ls '(type:photo + year:2005)'
```

The output will be something like:

```
ski/ holiday/ summer/ photoComposition-2005.jpg
```

where *ski*, *holiday* and *summer* are virtual directories and *photoComposition-2005.jpg* is a file. Suposing that the file *photoComposition-2005.jpg* has besides these two tags, the keywords *ski, holiday and summer*. Even if they weren't included in the query, they are still listed.

2.2.4 File rename

The renaming of files keeps the usual syntax and it can be used also for operations on tags.

Example: To rename the file *photo*1*.jpg* as $first_photo.jpg$, one can identify in a unique way the source and the destination by specifing the values for the special tags *path* and *filename* :

```
mv '(path:/directory/path + filename:photo1.jpg)' '(path:/directory/path + filename:first_photo.jpg)
```

or it can specify a set of files determined by combinations of tags, to be moved in another directory, and other tags to be assignated:

```
mv '(ski + holiday)' '(path:/pictures/alps/ + alps)'
```

2.2.5 Replace tags

This operation is done using the rename operation, but this time applied to tags only. With this operation, all the tags from the files depicted by the source query are removed, new tags being added by specifying them in the destination query. This will tell which are the tags that will replace the original ones.

Example: For all the files that have the *type:photo* and *author:stefania* tags defined, we want to delete all the other tags and transform the *type:photo* into *type:jpg*. This operation can be done as following:

```
mv '(type:photo + author:stefania + *)' '(type:jpg + author:stefania)'
```

2.2.6 File remove

In order to actually remove a file, you can use a query to point out the files based on tag information. All files resulted from the query will be (phisicaly) deleted along with their tag information from the DB.

2.2.7 File creation

If someone wants to create a file, than the special tags *path* and *filename* must pe specified. Also, to add tags at creation time, they must be specified in a conjunction query.

2.2.8 Add/Remove tags

This can be done by using the rename operation. Also, the tags can be specified at creation. Example: To add the tags *alps* and *ski* to the file */directory/path/photo1.jpg* :

```
mv '(path:/directory/path + filename:photo1.jpg)' '(alps + ski)'
```

3 Known limitations and issues

1. Because a complex query has a certain syntax, the following characters are reserved for describing the possible queries or the tag-associated values: (,), :, +, —, !.
2. File copy - how can files be copied and their tags be preserved without specifying them in the destination query? Do we need a special tool?
3. Query navigation - how can the query addition be realized if the path addition is done with "/" ?
4. Combined query sintax - how can the real path be specified without mentioning the special tags *path* and *filename*, in an easy way?
5. How should we treat multiple files that have the same name, but different paths?

4 HybFS Implementation

The implementation of the HybFS filesystem consists of three modules grouped togheter in one application. There is the frontend FUSE filesystem (also called "the hybfs core") and a backend comprising of a parser and the Berkley Database storage used for keeping the tag-files associations. The abstract flow of a request (or user operation) can be seen in the following diagram. The HybFS core interface makes the appropiate calls to the parser, the database and the underlying filesystem. The interface offered to the user processes remains unmodified.

1. A file system operation is issued (think of create, rename, unlink, etc.) and is passed to the HybFS filesystem.
2. The provided path is analyzed and validated by the query parser and the results are packed in an internal representation used afterwards to issue operations to the database backend.
3. The metadata from the main databases (as they are called in BDB) is accesed and/or modified.
4. The results are returned to the HybFS core.
5. Based on the results obtained, and if needed, the core interface issues a series of operations to the underlying filesystem(s).

4.1 Technologies

FUSE (File system in user space) is an abstraction layer that allows a fully functional file system to be written in user space, and allows regular users to mount file systems. It provides a simple and efficient API for rapid development.

Berkeley DB is an Open Source embedded database library that provides transaction-protected data management services to applications. Berkeley DB provides a simple function-call API for data access and management. It runs in the same address space as the application and all database operations happen inside the library. Multiple processes, or multiple threads in a single process, can all use the database at the same time as each uses the Berkeley DB library. Low-level services like locking, transaction logging, shared buffer management, memory management, and so on are all handled transparently by the library.

For the implementation of our modules we will also use:

- C - for the filesystem implementation
- C++ with (possibly) GTK+ for the policy management framework

4.2 Frontend

The filesystem can be mounted over a directory specified at mount time. When accesing the mount point, the user will be able to issue queries. In the first stage we only mirrored the original directory in the mount point. Next, we added support for BDB (Berkeley Database) and the implementation of file operations based on tags is still in progress.

4.3 Backend

4.3.1 Query Parser

(still) not implemented.

4.3.2 Interface with the Berkeley Database

The BDB is linked in the application as a library. For now we set only the main table that contains primary information about the tagged files. We store in a *hash table* information about the branch id, the file id (we choosed the inode number for this), the mode and the path name. The metadata directory is set in each mount point (branch), with the name **.hybfs** . All the files created by the DB enviroment and the databases reside there.

4.4 Policy Framework

The policy framework will provide an interface to define different tagging behaviours for the different mounted directories. This will allow the user to tag automatically files based on their types and the existing supported tagging modules (the first stage will be to extract information for MP3's or JPEG files).