HybFS: Adding multiple organizational views through a virtual overlay file system

***Abstract*—Traditional hierarchical file systems offer a single organizational view of the data, thus making file searching and browsing cumbersome in situations when only incomplete information is available. On the contrary, in a semantic file system, the navigation is based solely on the additional metadata associated with files, ignoring the original hierarchy. In this paper we suggest a non-intrusive method of adding semantic capabilities to the hierarchical file organization through a virtual overlay file system, thus making it accessible from any user application. This is done by tagging files with additional metadata, like tags that can have associated values. The user can organize files in a hierarchic way through the original file system operations but in the same time having access to advanced search capabilities and operations on multiple sets of files. The results of the search are listed through virtual directories which are generated at runtime.**

***Index Terms*—file systems, semantic desktop, tags**

I. Introduction

In a hierarchic file system, files can be categorized in only one way, by specifying their location. However, this approach is not well suited in all situations anymore, especially when dealing with a great number of files that are spread in a multitude of directories on the disk. Deciding how to organize files in the classic hierarchy is frustrating when there can be many possibilities to categorize the same file. Documents, music, images and movies are files that are always hard to manage. For example, a picture can be organized by the date and the place, by its content (it describes the city, or the landscape, or maybe it's a portrait), or its context (a long holiday, a weekend, or maybe a relaxing afternoon) but the user can choose only one file path to express all these attributes. To address this problem, applications for file indexing and semantic file systems are trying to offer navigation based on additional metadata associated with files But the hierarchical organization of a file system is not something to be eliminated because remains a logical and in the same time a unique way of identifying a file. One can still feel the need to organize the files in this way, but in a *full* semantic file system, the user has no idea how to identify the data. That is why the semantic file systems cannot replace the existing organization, but they can provide an additional extension to it.

In this paper we present a user-mode overlay file system that tries to combine the ideas used by previously semantic file systems in a non-intrusive method for file organization. We named our file system HybFS (Hybrid File System) because it offers both a semantic and a hierarchical view of the files. The file system can be mounted over a directory specified at mount time and, when accessing the mount point, the user can assign keywords to files and/or modify them, or issue search queries. Also, common tasks like extracting metadata directly from files or transferring the metadata between two HybFS mount points are accessible through an additional application. The file view provided by HybFS maintains the hierarchical representation because it is a good way of identifying files both uniquely and logically, and in the same time, when someone would want to create or move a file from our file system, will know exactly the original file organization from the underlying file system. Moreover, by not modifying the original file structure, this could be considered as an extra precaution in case something happens to the metadata storage to permit the user to continue working easily with the files.

This paper is organized as follows. The next section outlines the previous work on which we based our design while Section 3 presents the HybFS features. Section 4 deals with the design and implementation details and Section 5 describes the evaluation of our solution. In the end, we draw the conclusions and some future directions of development in Section 6.

II. Related Work

In this Section we will present some of the more important attempts of adding a semantic organization to the file storage. These solutions are not only at the file system level, but there are also desktop applications, each of these having its own advantages and drawbacks. Desktop applications are non-intrusive because most of them don't affect the file storage and they keep the additional information separate. Moreover, they can be accessible for the user, allowing advanced indexing configurations, but, in the case you change the application, all the work organizing the files is lost. Also, any other software from the system not only that will not be advantaged by this extension, but interfacing it with the desktop application will require changing parts of the software itself. Therefore, in parallel with the development of such desktop applications, efforts have been made to design file systems that integrate the same capabilities in an interface available from any user application.

A. Desktop Applications

Beagle [1] is a desktop search application that supports the indexing of file content and format and it also provides the capability to search inside IM conversations, mails, documents and web pages. It includes a GUI tool, an API for integration with other applications and a daemon for real-time indexing. For this purpose it uses the *inotify* interface provided by the Linux kernel to update in real time the information about the indexed files. Another well-known and used desktop application is *Spotlight* [2], a feature provided by Mac OS X. Spotlight it is based on a Metadata Server daemon that monitors the file system and responds to the search requests received from the clients. Besides the basic file information, the daemon can also index the file content based on additional plug-ins. Also, queries that support the boolean operators can be performed. Other semantic search tools, like Google Desktop Search, Windows Desktop Search or Yahoo! Desktop Search follow the same model and also parse and index the file content, offering advanced search capabilities.

B. Semantic file systems

The first concept file system that implemented the idea of semantic file systems was described by David Gifford et. al. [3] and afterwards led to a large amount of derivative work, including the semantic file systems and desktop applications recently developed. The main concept represented the adding of attribute-value pairs to the files from the hierarchical file system, attributes extracted from files with the help of *transducers*. The search result is returned as a directory created on the fly, also called *virtual directory*. Another interesting paradigm was implemented in the *Logic File System* [4], which treats paths like logic formulas. Also, files are indexed at creation, by using transducers. And, since tagging files with additional attributes doesn't explain the relationship between different set of files, the Linking File System (LiFS) [5] came with support for attributed links between files.

Other file systems that preceeded them, like Tagfs [6], Semfs [7], Orion [8], Insight [9], Tagsistant [10] start from the same ideas and don't take the original file path in consideration when performing queries, and not allowing the search request to be issued in an arbitrary directory from the file system (thus completly disconsidering the original hierarchical organization). Another solution, more oriented to a relational data storage approach was Windows Future Storage (WinFS) [11]. It was designed to allow users to find files using a structured query language, and it relies on a SQL Server layer. However, the real problem with semantic file systems is the fact that when creating a new file the user has no control of the real path.

A different approach, wich we also considered in our design, is presented by Deepak Garg et al. in WSFS [12]. They introduced the file path as a separate attribute-value pair. However, their implementation was incomplete and allowed only manual tagging by the user.

III. HybFS Semantics

With HybFS one can have access to tag information from the interface of a file system, specify paths for multiple directories and index the files through a separate console application with plugable modules. As an example, we can use the EXIF plugin for the images directory and the MP3 plugin for the music directory. However, for now the file indexing is not automated and we must specify the file or the directory to be parsed.

Additional attributes can be added to existent files or to the newly created ones. These are called *tags*, or *keywords* and can be extracted directly from the file content or added by the user through the file system interface. The tags can be simple, or can have an associated value that is used to granulate even more the description. In this case, the attribute represents the criteria of description and the value - the subcategory. If the tag doesn't receive an associated value, a default one,called *"null"*, will be assigned. Any combination of multiple tags, and/or tag-value pairs is seen as a virtual directory that has the same name as the search query itself. We call it *"virtual directory"* because it only exist as an abstraction exported by HybFS and created at runtime. An entry can be seen as a symbolic link to a file that match the current search pattern, or it can be a virtual directory that describes the other tags, or *tag:value* pairs assigned to the file, for further refining the query. In order for the user to have a hierarchic view, we define a directory with the name *"path:"* for the original mount point. This is desired for a better granularity. As an example, one can organize the pictures in directories based on the location where they were taken and refine the search with the aid of tags related to dates, people names, camera model and other.

The common file system operations are the same as in a normal file system but without support for symbolic and hard links. HybFS also supports operations like add, replace or remove for the file tags. The navigation in HybFS resembles the navigation through a normal hierarchic file system, except that the file path can be also based on the tag information associated with the files. From the application point of view, the result of a query is seen as a directory.

A. Query syntax

A query can be composed from:

- A tag, or a tag and value pair: $(picture)$, $(picture : autumn)$
- A conjunction, disjunction or negation of tags: $(picture : autumn + myself)$, $(picture : autumn|picture : winter)$ or $(!myself)$
- One or many conjunctions, disjunctions or negations of queries: $(q_1 + q_2)$, $(q_1 \mid q_2)$ or $(!q_1)$, where q_1 and q_2 are simple or composed queries.

B. HybFS operations

We describe the common HybFS features in terms of shell commands because they are user oriented and well known.

cd: The equivalent of a change directory is a *refine query*. The resulted query is applied to the files that match the current query, or, if the path has a real component, to the files from the resulted path.

ls: This command will list all the files that match the conjunction from the current query and the specified query, or, if a real path is specified, all the files that are in that path and match the query. When listing the root directory, we will see all the tags and values from the file system, and the special directory *path:*. Also, virtual folders to refine the results of the navigation are added. Thus we solve one drawback of a full semantic file system in which one doesn't receive any suggestions when navigating through the virtual directories as it would have received in a hierarchical file system. For example, when listing all the pictures that have been taken in the year "2008" and are in the real directory "My Pictures", or in its sub-directories, virtual directories are created from additional tags of the files already found that are not included in our query:

```
$ls 'path:/My Pictures/(year:2008)'
IMG_6577.JPG    IMG_6590.JPG
IMG_6578.JPG    IMG_6639.JPG
IMG_6579.JPG    IMG_6662.JPG
(picture:autumn)
(picture:winter)
```

rm: This keeps the original syntax: it will delete the file uniquely determined by the specified path. Also, it can be used to delete *sets* of files that match certain queries, by specifying virtual directories.

mv: The move operation has a special syntax, because it can be used for changing the set of tags for a file, or a set of files. The renaming of files keeps the original syntax. If the destination file path contains a virtual directory, then it will try to do a tag operation based on the queries that form the virtual directory path. Also, the queries that specify the new tags to be assigned must be based on conjunctions only. This is happening because a different syntax will be too ambiguous. If there is more than one query in the path, then the operations will happen in the specified order. The syntax of the destination query is:

- $(tag_0 + tag_1 + tag_2 : value_2 + ...)$: all the tags for the files depicted by the source query and/or directory are removed and the tags $tag_0, tag_1, ...$ are added. If a tag doesn't have a value specified, then the "null" value will be assigned.
- $(|tag_0 + tag_1 + tag_2 : value_2 + ...)$ All the tags for the files resulted from the source query and/or directory are kept, the new tags are added.
- $(!tag_0 + tag_1 : value_1 + ...)$ It removes all the pairs $tag_i : value_i$ for the files resulted from the source query. If the value is not specified, than it removes the pair, indifferent of the tag value.

For example, the command $mv\ 'q_0/\ *'\ \ '/path : /dir\_path/q_1'$ will execute the operation on tags described by q_1 for the files that match the query q_0 and, in the same time, these files will be moved in the directory "$/path : /dir\_path/$".

cp: The copy operation will not assign the existent tags of the source file to the destination file. However, tag assignment can be explicitly specified, by adding to the destination path the conjunction query that contains them. To properly copy a file in the HybFS file system, or between two HybFS mount points, we designed a similar tool to *cp*.

Other file system common operations that are applied to files can be valid only when the path has a real component (the path can have a virtual component also, when the files are the result of a query). The *create* operation is somehow different, because it will interpret the virtual component of the path as a request to add new tags to the new file. Therefore, the query must be formed only from conjunctions, to prevent an ambiguous request.

IV. Design and Implementation

The implementation of HybFS consists of three modules as described in Figure 1: the user-space file system, a library designed with the purpose to offer an uniform interface to the metadata storage and an application that allows loading of multiple plug-ins for extracting metadata from files, getting statistics about tags for testing purposes and transferring files between two HybFS mount points, together with their metadata. The plug-in support is represented through a generic interface that can be extended by each new plug-in sub-module. For further developing purposes, the library adds support for multiple instances of the back-end interface, called *'virtual directory branches'*, each one of them having its own database connection. However, for now, this can be useful only for the HybFS application.

A. Front-end Module

The front-end is represented by the HybFS file system interface. We implemented our file system with the FUSE [13] toolkit for user-level file systems in Linux, which is in turn based on the Virtual File System Layer (VFS). In this way, we structured HybFS as a layer of extended-content around an already existent location on the file system. The HybFS file system application implements the common file system operations needed to provide basic functionality. Also, it is responsible for initializing the back-end interface for the mounted directory, parsing the paths and passing the results to it. The abstract flow of a simplified request is described as follows:

1) A file system operation is issued (e.g. create, rename, unlink) by the user's application and it is passed by the VFS to the FUSE file system driver and which in turn calls the appropriate HybFS function.
2) In HybFS, the provided file path is parsed and the resulted queries and the real file path (if any) are packed in an internal representation used afterwards to issue operations to the metadata database.
3) In the case of a tag operation or a query, the metadata from the database is accessed and, possibly modified.
4) The results are returned to the HybFS core.

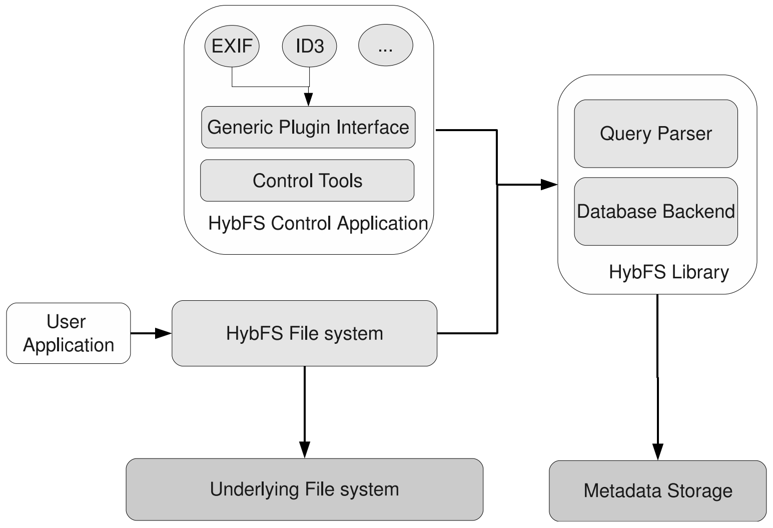

Fig. 1. HybFS Design

5) Based on the results obtained, and if needed, the HybFS interface passes the operation to the underlying file system.

B. Backend Module

The backend module is represented by a library used to access the metadata database from the file system interface and our application. This provides a uniform interface to the query parsing and the database access internal methods, allowing the future development of other metadata indexing solutions.

For indexing the semantic attributes of the files, we use a relational database. This allows any search query to be directly mapped on a SQL query. The database is implemented using Sqlite, which is linked in the application as a library. The database contains three tables with information about the tagged files. The first table contains the file inode number, the file mode and the real path relative to the mounted directory. Also we keep all the tags assigned for a file in a separate field, for issuing search queries faster. The second table is for storing the tag and value pairs, together with a unique id and the last one keeps the association between a file id and a tag:value id. The last two tables can be used for tag statistics from the HybFS Control Application. For simple tags (that don't have a value), we assign the "null" value. The metadata directory is set in each mount point, with the name ".hybfs". The main database is kept in the file ".hybfs_main.db", thus all the tables are in the same file.

C. HybFS Control Application

The HybFS Control application provides an interface to define tagging behaviors for different mounted directories. This will allow the user to tag automatically files based on their types and the existing supported tagging modules. For now we support tag extraction for MP3's and JPEG files. The application allows specifing multiple HybFS mount points and for each mount point a different set of plugins can be loaded. For example, if we have the /fs1 and /fs2 as the directories in which HybFS was mounted, we can load the MP3 plugin for /fs1 and both the MP3 and EXIF plugins for /fs2. When indexing a directory from /fs1 only the mp3 files are parsed and their information loaded into the appropriate database while from a directory that is found in /fs2 both mp3 and picture files are processed.

The application can also be used for special file operations like copy and move. When files are copied between different HybFS mount points, their tags are also transfered between the two databases. Even if there wasn't any HybFS file system mounted for a specific location, that location can be defined as a mount point in the application and a new database is created in order to permit copy/move operations to/from the new defined location. This allows the transfer of files together with their additional tags. The operation needed for this implies the defining of the directory where the files will be copied as a mount point using the HybFS Control Application on both the source and the destination. When defining an additional directory as a mount point, the application will initialize the database storage for tagging information, if it is not defined already.

V. Evaluation

HybFS is an overlay file system implemented in user-space, meaning that will imply an aditional overhead for file operations. This overhead is also due to queries made to the metadata storage. [11] and [8] use an external database server for indexing the file information. Unlike them we chose Sqlite3, because is an embeddable database and it removes the process of running a database manager all the time.

We evaluated HybFS in terms of usability and we measured the average time for locating and copy files which are identified based on a criteria. There are cases in which

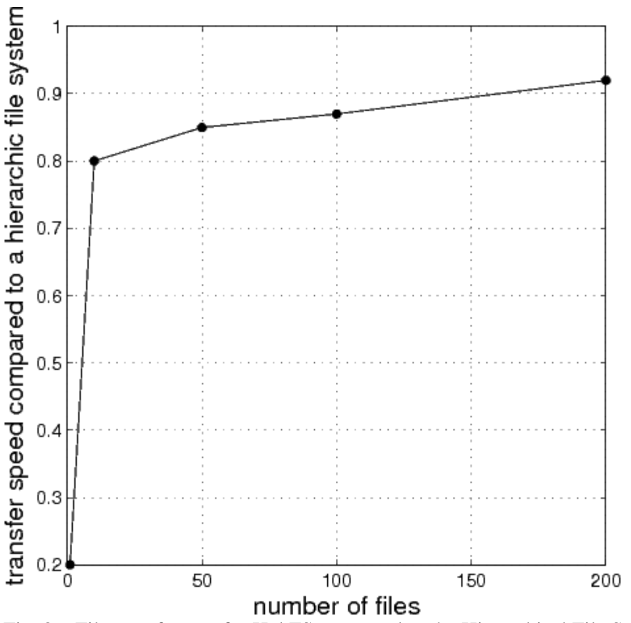

Fig. 2. File transfer rate for HybFS compared to the Hierarchical File System beneath.

the tag navigation is not necessary. For instance, there are files that are well identified by their name and the directory structure in which they are found is very simple and logic. This usually happens when dealing with file types that are weakly represented or all the files are well grouped in very few locations. But even for these files, a certain improvement can be obtained by combining the hierarchical approach with the semantic one. If the user knows where certain types of files are located, he can navigate to that location in the classic way and then use tags to provide a more insightful description. This is the case for a well organized folder structure, where the user is presented with very strong clues when browsing through the file structure. But the storage capacity has increased recently and the folder structure exceeds easily 8-9 levels of hierarchy, seldom with repetitive folder names being present in the path. Moreover, many files don't have a suggestive name in order to be found easily just by reading it and not opening the file.

To exemplify the usability of HybFS we will present the following use case. We start with the assumption that we have a file system whose directory structure is 6 levels deep. Among the numerous files of various types, there are also around 500 photos. These pictures are already organized by the place where they were shot and the people involved. But there are also some files that are first organized based on the location and as we go deeper into the folder structure based on the person who made them, while others are organized by the date they were shot first, and second based on location. Not to mention that there can be cases where some files are organized based on location, date and location again, maybe files that were transfered in a hurry from the camera. This is how a normal hierarchical file system usually looks like.

These files can be tagged automatically based on EXIF information (year, month, camera model, description) with

TABLE I
AVERAGE TIMES FOR A FILE COPY SCENARIO.

Nr files	HybFS	Hierarchic FS
1	0.49s	0.09s
10	0.72s	0.65s
55	4.23	3.82s
100	12.22	10.81s
200	25.13	22.87

HybFS control application. More than that, for certain groups of files, tags were added manually by using the *cp* command with the appropriate syntax. In order to ease this process HybFS permits to add/remove tags from groups of files from a certain directory or obtained from a query.

Using these tags we can improve the file searching process a lot, especially when knowing the date, location, camera model or the characters that have something to do with the file. For example, one wants to search for a picture that he or she knows that it was shot in France, more exactly in Paris near Sena river and it shows Bob as the central character. In the folder structure in the *photos/* directory there is a folder *France/*. For this, by using HybFS the following steps could be taken (but they are not necessary, other combinations of steps can exist):

1) browse in a real directory: In order to find this picture, one can browse to *photos/France/*

2) list the files that have a combination of tags: Follow the next scenario, represented as shell commands, but also followed from any file browser:

```
$cd path:/photos/France/
$ls '(sena+character:bob)'
(camera:Canon_DIGITAL_IXUS_75)
(paris:null)
(type:image)
(month:april)
Paris/Bob/Sena.JPG
(year:2008)
```

3) do something with the result: Maybe the files need to be moved somewhere else. This can be done with the *mv '(sena+character:bob)'/* path:/temp* command.

The classic scenario for this could involve more than three commands (or steps) if the files are scattered in several directories, a simple action like 'move all the files that have the character Bob and are taken on Sena'. And if the files are not named accordingly you would certainly have to open them. For example if someone will want to search for all the photos that have flowers in them all that has to do is to follow the next steps:

```
$cd '(flowers)'
$ls
 photos/19_may/Vannes/Vannes&Auray2.jpg
 photos/2007_contry_side/P1010078.JPG
 photos/France/Paris/Anca/Flori.JPG
 photos/France/Paris/Dan/Mar/1.JPG
 (location:vannes)
 (location:dersca)
 (location:paris)
 (location:marseille)
 (camera:C750UZ)
 (author:dan)
 (color:blue)
 (color:yellow)
```

These are all the images with flowers alongside their other tags. After that, they can be further refined: to see the pictures that contain blue flowers, a new filter can be added:

```
$ls '(color:blue)'
 photos/France/Paris/Dan/Mar/1.JPG
 photos/France/Paris/Anca/Flori.JPG
 (author:dan)
 (location:marseille)
 (location:paris)
```

In a normal file system, this shurely could have involved more than one browsing step.

HybFS was also tested in terms of *efficiency*. We measured the time for copying several files identified through a query and we compared it to the time for copying the same number of files in a hierarchical file system (the average size of a file is 1.8 MB). For the time measurement we used the *time* system command and we present the average values in Table I.

As we can see, the overhead in terms of execution time is around 10%-15% for HybFS when having over 1000 files indexed in the database. There is also an isolated case where the overhead is over 500%, but this is only for copying one file. The average time for the execution of a query is between 0.1 and 1 second, time that considerably affects the execution time in the case of a single file copy. Moreover, to uniquely identify a single file, the query that was issued was a more complex one. But this is considered acceptable in terms of user efficiency and knowing that by using tags the operation is practically a search through the whole file system. We present in Figure 2 the relative HybFS speed of file transfer(copy). The transfer speed on HybFS is compared to the speed of the underlying hierarchical file system, which we consider to be the maximum value of 1.0. We can notice that the transfer speed of HybFS starts at about 0.2 the speed of the system beneath for 1 file and it stabilizes at around 0.9 as the number of files increases.

Using HybFS is not a matter of file transfer throughput, but one of a solid search method that will improve the user's browsing experience.

VI. Conclusions

In this paper we presented an virtual overlay file system that provides semantic capabilitites to normal hierarchic file systems, thus allowing advanced file searching and tagging from any user application. All the additional metadata is kept sepparate and with this aproach we don't modify the original file structure and we allow reverting to the old view at any time.

For now, to use the HybFS Control Application, the user must run the application and issue commands from the console. Further work is being done to split the application in two modules: a deamon that runs in background and takes care of the module loading and file parsing and a set of tools so that one can have access to special HybFS functions from anywhere. Also the application can be improved by automating the parsing of files and adding a possible integration with indexing solutions used by popular applications. How the solution was tested on Linux, the next steps will be porting it to other operating systems. We plan to replace of the FUSE interface with a similar solution for Windows and Mac OS X. In the end, allowing collaborative file tagging and searching by integrating HybFS with a peer-to-peer file system will be the next challenge.

References

[1] "Beagle," http://beagle-project.org/Main_Page.

[2] "Spotlight," http://en.wikipedia.org/wiki/Spotlight_(software).

[3] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'toole, "Semantic file systems," in *Communications of the ACM*, 1991, pp. 16–25.

[4] Y. Padioleau and O. Ridoux, "A logic file system," in *USENIX Annual Technical Conference*, 2003.

[5] A. Ames, C. Maltzahn, N. Bobb, E. Miller, S. Brandt, A. Neeman, A. Hiatt, and D. Tuteja, "Richer File System Metadata Using Links and Attributes," in *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on*, 2005, pp. 49–60.

[6] S. Bloehdorn, "Tagfs - tag semantics for hierarchical file systems," in *In: Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06*, 2006, pp. 6–8.

[7] P. Mohan, S. Venkateswaran, M. Raghuraman, A. Siromoney, and I. Chennai, "SemFS: A Semantic approach to File Systems."

[8] F. S. A. Joshi and S. Todwal, "Orion File System : File-level Host-based Virtualization."

[9] D. Ingram, " Insight: A semantic file system."

[10] "Tagsistant," http://www.tagsistant.net/.

[11] "WinFS," http://en.wikipedia.org/wiki/Winfs.

[12] S. P. D. Garg, V. Mehta and M. D. Rosa, "Writable Semantic File System."

[13] "FUSE: Filesystem in Userspace ," http://fuse.sourceforge.net/.