GFS: a Graph-based File System Enhanced with Semantic Features

Daniele Di Sarli
University of Pisa
Pisa, Italy
d.disarli@studenti.unipi.it

Filippo Geraci*
Istituto di Informatica e Telematica, CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
filippo.geraci@iit.cnr.it

ABSTRACT

Organizing documents in the file system is one of the most tedious and thorny tasks for most computer users. Taxonomies based on hand made directory hierarchies still remain the only possible alternative for most small and medium enterprises, public administrations and individual users. However, both the limitations of the hierarchical organization of file systems and the difficulty of maintaining the coherence within the taxonomy have raised the need for more scalable and effective approaches.

Desktop searching applications provide proprietary interfaces that enable content-based searching at the cost of having no control on the indexing and ranking of results. Semantic file systems, instead, leave users the freedom to manage the taxonomy according to their specific needs, but lose the standard file system features.

In this paper we describe GFS (graph-based file system) a new hybrid file system that extends the standard hierarchical organization of files with semantic features. GFS allows the user to nest semantic spaces inside the directory hierarchy leaving unaltered system folders. Semantic spaces allow customized file tagging and leverage on browsing to guide file searching.

Since GFS does not change the low-level interface to interact with file systems, users can continue to use their favorite file managers to interact with it. Moreover, no changes are required to integrate the semantic features in proprietary software.

Categories and Subject Descriptors

D.4.2 [**Storage management**]: Storage hierarchies

Keywords

Semantic browsing, file tagging, user experience.

* Corresponding author

1. INTRODUCTION

Handmade directory hierarchies still remain the only method to classify documents for most computer users. Surprisingly, even public administrations as well as small and medium enterprises rely on manual classification. Once a new administrative task is started, the secretariat staff creates a new folder with a self-explaining name inside a directory tree. The directory path often consists of a base path (the hierarchy root) and a list of subdirectories representing a set of tags describing the inner documents. For example, the path /documents/contracts/2017/supply/company_XYZ/signed/ refers to the folder of the signed contract stipulated with the company XYZ for a supply service in 2017. The hierarchy root is located in the physical directory /documents.

The disadvantages of this organization are evident since, as observed in [7], the user has to deal with complex information management problems in order to maintain consistency within the taxonomy, and, in turn, to be able to locate files.

These problems are further complicated by the severe limitations of the manual hierarchical organization of files [13]. In fact, since adding a new tag corresponds to push a file down one level in the hierarchy, the number of tags that can be used in practice for a single file is very limited. This, in turn, drives the user to create additional meta-categories that are the result of merging together subsets of tags. For example one can be induced to create the meta-category 2016-17 for those documents that are valid across both the years. This, however, causes the documents in this directory not to be shown neither in the directory of 2016 nor in that of 2017. Another important limitation is that tags typically belong to different categories (i.e. document type, period, etc.). The rigidity of the hierarchical organization of the file systems forces the user to nest these categories. In absence of a rigid rule about the precedence order among categories, this can cause an inconsistent organization of different branches of the same hierarchy.

Despite their limits, file systems have huge advantages. They are natively present with no extra costs or struggling with installation in every desktop operating system. Moreover, OSs expose easy and convenient APIs (Application programmable interface) that enable applications to control the file system hiding the underlying low-level details. As a result, users can interact with the file system by means of the same standard interface either within or outside applications. In turn, this fact has an impressive positive impact. In fact, a common interface opens to the possibility of interoperating on the same file hierarchy among different applications without requiring the user (and even the applications) to be aware of it or to do any action to enable sharing.

Aimed at overcoming the limits of standard hierarchical file systems several alternatives have been proposed in the literature (We will discuss them in more detail in section 2). Desktop search applications are stand-alone software that enables keyword-based searching. Their objective is not that of improving the file system organization but that of easing the retrieval process making file location de facto not important.

Semantic file systems (SFS) goal is that of replacing the position-based with an associative-based access to files. In their early stage these file systems extended the API adding new system calls for controlling tags and to perform searching. This required them to be endowed with an ad-hoc browsing application. More recently SFSs strategy has evolved leaving the system APIs unaltered changing only the behavior of calls as to provide associative access to the files. The advantage of this latter strategy is that standard applications directly inherit the new associative capabilities [8]. On the other hand, however, semantic file systems lose the original position-based file access.

In this paper we try to address the question whether it is possible to extend standard file systems adding extra semantic features without altering the API or not.

Our key idea is that standard and semantic directories coexist in the same tree structure and the file system is provided with a criterion to decide the directory type (either semantic or standard). Consequently, it is possible to dynamically change the API behavior according to the context. Our hybrid approach takes advantage of the benefits of semantics without sacrificing any of the advantages of the classic hierarchical file systems. In fact, in the absence of semantic directories, our file system reduces to a standard one. Each semantic directory behaves as a stand-alone semantic file system with its own namespace and set of tags. This allows overcoming the limitations due to a single namespace for the whole semantic file system. Lastly, our approach can be extended with new directory types simply extending the classification criterion for the directory type and adding the new semantic to the API.

2. RELATED WORK

Implementing a new file system is a complex task. In fact, it requires writing procedures in kernel space as well as controlling all low-level hardware details. This difficulty has led researchers and companies to develop stand alone searching software in place of semantic file systems.

Desktop search applications directly designed by OSs producers (Apple's Spotlight [3], Linux KDE Baloo [6], and Windows Desktop Search [11]) are among the most successful solutions because of their native integration with the underlying operating system. Besides general-purpose desktop search software, some applications for specific problems have been proposed. Mendeley [15], for example, is a popular tool for managing and sharing pdf articles.

All these solutions, however, have in common that the user is obliged to use an ad-hoc interface to locate files. Retrieval is based on keyword oriented searching while browsing is not supported. Moreover, third-party applications have to implement ad-hoc interfaces to access the searching features or, more often, have no access at all.

The advent of FUSE (File system in user space) had the effect of a resurgence of research about pure semantic file systems. According to [12] developing in user space produces a consistent blowup of performance for a single small writing operation, but the gap becomes negligible with increasing data transfers. Consequently, FUSE does not change the file system user experience.

Only few approaches, however, exploit the standard POSIX API for managing semantics allowing users to keep using their favorite file browser. SFS [8] is an early attempt in that sense. This file system consists of two components: an indexer that extracts semantic tags from files and the driver that exports the POSIX API. In comparison with a standard driver, the only modification is in the *readdir ()* system call. Besides standard paths, this call can also accept extended paths where a list of tags is specified. The resulting *virtual directory* contains only the subset of documents matching all the tags in the subtree of the specified path. Editing a path, however, was a common practice in the age of command line interfaces, but it is impractical nowadays with graphical interfaces.

In [9] the authors argue that the user may want to control which portion of file system must have a standard behavior and which must be semantic. In order to enable this option, they extend SFS with the concept of virtual mount point (namely a directory that is the root of an SFS instance).

TagFS [4] is the most similar to our approach and it is the only one that enables a manual control of the taxonomy. As in our solution, the creation of a directory is equivalent to the creation of a new label and tagging a file is controlled with the file copy operation. There are, however, some important differences. Firstly: TagFS semantic features substitute the canonical behavior, thus losing the standard capabilities; moreover: the entire file system share the same namespace, thus no two files can have the same name. Another important difference is that TagFS implicitly organizes tags as a clique (it is always possible to move from a tag to another) while our solution uses a series of editable ego graphs. This feature is important to control the number of visualized items by the directory listing when the number of tags increases over few units (see section 3.4).

Most effort has been spent in solutions that extend file system API with metadata that enables searching.

In [1] the authors propose LiFS a hierarchical file system extended with application customizable attributes in the form of (key=value). LiFS enables also links among files (for example a document can be linked to the appropriate viewer). However, these links are not used to improve the user navigation experience.

In [14] the authors propose a distributed index that speeds up metadata searching in the context of high-throughput computing.

In [2] the authors observe that the POSIX interface for user metadata storage has become a major bottleneck for very large file systems. The authors also criticize the hierarchical organization and propose a graph-based ad-hoc file system to store metadata as opposed to relational databases. File identification and attribute retrieval is performed using a query language interface. Although [2] shares with us the idea of using a graph instead of a tree to organize a file system, their work is tailored on data access performances and not on the user experience.

Sometimes semantic file systems have been used for special purposes. In [10] the authors face the problem of keeping version history of all files introducing *Sedar*: a file system for deep archival. *Sedar* uses semantic metadata to store semantically similar documents close to each other on the disk. This enables a faster access to related documents as well as the possibility of using locality to compress data reducing the storage consumption.

Proposing SIL (Semantic instead of Location) [5] the authors face the problem of integration among data management software participating in the business process from a different point of view. Their idea is that of replacing the path-based hierarchical location with a semantic one. In short, the path is not seen as an ordered list of directories but as a set of attributes. As a result, accessing a file does no longer require remembering the order of the attributes. Similarly to our approach SIL maps directories into attributes. However, SIL still requires knowing all the tags to find a file. Moreover, the file system behavior is uniform both in the user data directories and in the system directories. This, in turn, opens to security issues.

3. GFS DESCRIPTION

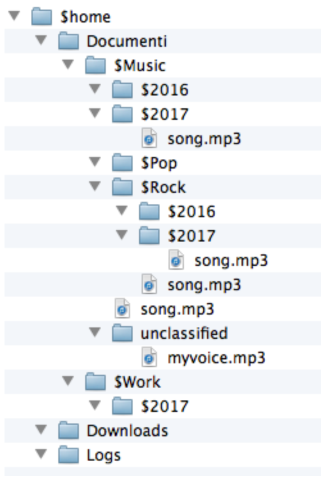
At first sight our approach could be seen as a mix of HAC [9] and TagFS [4]. In fact, as in [9] our idea is that, if it is to be used in practice, a file system must provide both: the standard features for system folders and a series of separate semantic spaces for user's collection of data. As in [4], users must be able to build their own taxonomies (which in our case are not organized as a big clique) and tag files accordingly. Moreover, to relieve the user effort of manual tagging, the file system must provide natural methods to assign multiple tags as well as tagging multiple files at once. Since current file managers are all designed for browsing, this should be the retrieval paradigm both inside and outside the semantic spaces.

GFS can be seen as a standard file system where in every location it is possible to create two types of directory: standard and semantic. Standard directories have their own namespace and behave as in traditional file systems independently from the type of their direct ancestor. A semantic directory having a standard directory as a direct ancestor defines a new *semantic space* with its own namespace, while a semantic directory that has a semantic direct ancestor defines a new tag. We call the root of a semantic space *entry point*.

A semantic space can be modeled as a direct connected ego graph where the entry point is the center and the tags are the other nodes. Once a new tag is created two edges from and to the entry point are created. In addition the new tag is doubly linked with all the tags in the path to the entry point. Creating a standard directory inside a semantic space causes the exit from this space. This enables the possibility of alternating semantic spaces with standard hierarchies. A usage example of this mechanism could be the case of a user who wants their home directory to be semantic but they also want a separate namespace for music, one for their working documents and a standard behavior for log files and downloads (see figure 1).

3.1 Identification of the directory type

Evaluating the directory type is the most used subroutine inside the GFS business logic. It is repeated at least once for every call to the file system API, thus an efficient mechanism for this task is crucial.



```
▼ $home
  ▼ Documenti
    ▼ $Music
      ▼ $2016
      ▼ $2017
          song.mp3
      ▼ $Pop
      ▼ $Rock
        ▼ $2016
        ▼ $2017
            song.mp3
          song.mp3
        song.mp3
      ▼ unclassified
          myvoice.mp3
    ▼ $Work
      ▼ $2017
  ▼ Downloads
  ▼ Logs
```

Figure 1. A sample tree view of a semantic home directory that mixes standard directories with semantic spaces for: music and working documents.

Moreover, the directory classification cannot change the POSIX interface. To do this, GFS encodes the directory type directly in its name requiring semantic directories to start with a special symbol. Although this introduces a little limitation on the user's choice of the directory names, it is a commonly accepted practice (for example: filenames beginning with dot are hidden in the directory listing of the UNIX ls command, additional metadata files often begins with dot followed by underscore "._") that has the advantage that it does not require accessing to extra information. In our file system we left the prefix for semantic directories configurable. For the sake of explanation in this paper we use the dollar ($) symbol.

3.2 Graph management

Since entry points and tags are special types of directories, they are manipulated by means of the standard system calls for directory management. *Mkdir ()* is used to initialize a new semantic space, to add new nodes or to add new edges while *unlink ()* is used to remove nodes, edges or the entire semantic space.

3.2.1 mkdir ()

Besides standard directories, this call is used to create semantic directories. Specifying whether the semantic directory is an entry point or a tag is not necessary because this information is derived from the direct ancestor. In fact, since semantic spaces cannot intersect to each other, an entry point always has a standard parent directory while tags have a semantic direct ancestor. Contrasted with standard directories, creating an entry point does not produce appreciable differences from the user point of view: a new node is created and linked to its direct ancestor, then the namespace is initialized.

Creating a tag, *mkdir ()* verifies whether the corresponding node already exists and creates it if necessary. Then the new tag is connected to the graph adding edges from all its direct semantic ancestors. Finally, in order to maintain the ego structure of the semantic space, a link from the corresponding entry point is created as well.

3.2.2 *unlink ()*

The unlink call receives in input the absolute path of the directory to remove (target) and its behavior depends on the type of the direct ancestor. If the ancestor is non-semantic, *unlink ()* has a standard behavior independently from the type of the target. Otherwise, if the ancestor is a tag, the link between the tag and the target is removed, but the directory is not erased. This leaves it accessible from other paths, thus, in this case the target is not required to be empty. Finally, if the ancestor is an entry point all the references to the target are removed and the directory is erased.

3.3 File tagging

GFS uses two equivalent methods to tag files and directories: linking and copying. Linking is done by means of the *link ()* system call. To speed up tagging, GFS assigns all the tags of the destination path. As a further simplification, multiple assignments of the same tag do not raise an error but are silently ignored.

Copying requires some extra caveats. This operation, in fact, is not atomic but it is the effect of a series of system calls that cause the impossibility for the file system to be aware that they correspond to a file copy. In particular, copying:

1) the target file is created or opened in writing mode,
2) the *truncate ()* is called to set the file size to 0,
3) a series of writing operations copy the content from the source file into the destination,
4) the target file is closed.

This mechanism has two problems: firstly, since the source and destination file are in reality the same file, the *truncate ()* would affect the source file destroying its content; secondly, the series of unnecessary physical writing would cause an increase of latency of the tagging proportional to the file length. We address both these issues using virtualization. When a file in a semantic space is opened in writing mode a virtual file (called *ghost file* is created. All the subsequent operations are not applied to the physical file, but the modifications are recorded in the ghost file. Concurrent accesses to the same file are diverted to the ghost file. When the file is closed all the modifications are applied to the physical file and the ghost file is removed.

Notice that the series of *write ()* calls due to file copy do not modify the source file and, thus, they are not stored in the ghost file. This, in turn, avoids unnecessary writes on the disk and the consequent time and resources consumption.

3.4 Visualization

The size of a semantic directory can quickly increase with its usage. In particular entry points list all files, directories and tags of a semantic space. Moreover, it is very likely for a semantic space to contain cycles that, in turn, can generate either infinitely long paths or an infinite number of alternatives to reach the same file. Until now, dealing with these problems has been left to file managers. However, if not well addressed, visualization can cause a very poor user experience and, consequently, discourage the user from using semantic file systems.

We deal with the problem of the directory size by controlling the order in which the *readdir ()* system call returns the elements of a semantic directory. We first return the list of tags, then files in lexicographic order, finally standard directories in lexicographic order. Pushing directories on the bottom of the listing has the benefit of keeping them visually separated from tags (Note that directories and tags have the same icon). Sorting tags is more complex. The lexicographic order eases searching but spreads tags of the same category (i.e. classifying music the genres would be mixed with the artist names). Besides lexicographic ordering we provide a heuristic that attempts to mitigate tag spreading. We hypothesize that two tags of the same category are unlikely to be directly connected. We further observe that the entry point-centered ego structure of semantic spaces causes the shortest path between two nodes to have length 1 when the corresponding tags are connected, 2 otherwise. Let $T = \{t_1, t_2, \dots, t_n\}$ be the lexicographically ordered list of tags in a semantic space, given a tag t, let $\phi(T, t)$ be the sorted list of tags of T at distance 2 from t. Under our hypothesis the list t, $\phi(T, t)$ is likely to be a category, thus its elements should be visualized consecutively. Our heuristic works as follow. Let L be a new initialized list, iteratively: removes from T the first element $t = t_{\min(i)}$ and append it to L. Then extract from T the elements $\phi(T, t)$ and append them to L. The procedure stops once that T becomes empty. The ordering of L is used to display the tags.

GFS uses a very simple and effective method to prevent users from following infinite paths during browsing: visualizing a semantic directory, tags already present in the path are not shown even if the link between the corresponding nodes exists. In general, paths with multiple instances of the same tag are considered as not valid, thus, attempting to access them, the system call returns the *ENOENT* error.

3.5 Tagging shortcuts

The cost of allowing a total control on tagging is that of leaving all the burden of manually classifying each single file to the user. Without ad-hoc mechanisms that simplify this task, manual tagging can quickly become impractical and, in turn, makes the semantic features unusable. We designed several shortcuts that help users to quickly build or replicate their taxonomies and enable fast multiple tagging of files.

3.5.1 *Enable recursive copies*

Software for recursive copy of directory trees often uses a depth-first approach. Creating the taxonomy structure, however, this approach can generate a series of EEXIST errors due to the fact that the tag has already been created in a deeper position. We deal with this problem relaxing the behavior of *mkdir ()* when involves a semantic space. In this case, if a directory already exists, the system call does not do anything.

A similar problem can arise copying files. In fact, since the same file can have multiple paths, the recursive copy could erroneously attempt to copy a file on itself (i.e. applying the same tag several times). In this case, however, the mechanism described in section 3.3 prevents the file to be truncated and correctly manages the copy.

3.5.2 *Enable multiple tagging*

The easiest method to apply multiple tags at once is that to label files with all the tags of the semantic path where they are copied. This simple mechanism, however, is not powerful enough when dealing with large amounts of files. We control multiple files tagging by means of the *rename ()* system call. In particular GFS enables a standard directory to become semantic and vice versa.

In the first case all the files are moved in the semantic space and are tagged according to the destination path. The procedure returns an EEXIST error in case of clashes in the namespace and no file is moved.

Converting a semantic directory into standard poses some issues. According to the expected behavior:

1) the target should be standard and should contain all the files of the source,
2) the internal tags should be destroyed and,
3) the source directory should be completely deleted.

However, if the source is a tag, its removal does not imply the removal of the inner files that remain accessible from other paths. On the other hand, if files are not removed the *rename ()* would have the effect of copying files instead of moving them. We found that ensuring the above three properties (thus removing the files from the semantic space) is more intuitive than producing copies of the files even if this means that the *rename ()* affects the entire semantic space.

4. CONCLUSIONS

Organizing files is a frustrating and ubiquitous task that discourages most computer users. Accurate taxonomies enable fast retrieval of documents at the cost of a huge effort of building and keeping them tidy. Misclassification of a file is not much better than its accidental deletion. Content-based searching applications can be helpful for general users, even if the lack of any control on the indexing and ranking mechanisms makes them of little use when the volume of data increases.

In this paper we introduced GFS, a new hybrid file system that overcomes the limitations of the tree-based organization of file systems allowing semantic spaces to be mixed with the classic hierarchical structure. Our file system does not require any modifications to the standard POSIX interface and, thus, leaves users to control all details by means of their favorite file browser.

Although GFS shares some ideas with other proposals, previous approaches suffer from a lack of usability that caused their limited practical use. Designing GFS we focused on this aspect conveying the idea that file systems cannot delegate the user experience to file managers. The mechanism to create or assign multiple tags at once, that for multiple file simultaneous tagging, and a sensible ordering of the directory listing, are just a few examples in this direction.

5. REFERENCES

[1] Sasha Ames, Nikhil Bobb, Kevin M Greenan, Owen S Hofmann, Mark W Storer, Carlos Maltzahn, Ethan L Miller, and Scott A Brandt. 2006. LiFS: An attribute-rich File system for storage class memories. *In Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies.*

[2] Sasha Ames, Maya Gokhale, and Carlos Maltzahn. 2013. QMDS: a File system metadata management service supporting a graph data model-based query language. *International Journal of Parallel, Emergent and Distributed Systems 28, 2 (2013), 159-183.*

[3] MAC Apple. 2012. OS X Spotlight. (2012).

[4] Stephan Bloehdorn, Olaf G□orlitz, Simon Schenk, Max V□olkel, and others. 2006. Tagfs-tag semantics for hierarchical File systems. *In Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06), Graz, Austria, Vol. 8.*

[5] Oliver Eck and Dirk Schaefer. 2011. A semantic File system for integrated product data management. *Advanced Engineering Informatics 25, 2 (2011), 177-184.*

[6] KDE e.V. 2016. KDE Baloo. (2016).

[7] Sebastian Faubel and Christian Kuschel. 2008. Towards semantic File system interfaces. *In Proceedings of the 2007 International Conference on Posters and Demonstrations-Volume 401. CEURWS. org, 20-21.*

[8] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. 1991. Semantic File Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*. ACM, New York, NY, USA, 16-25. http://dx.doi.org/10.1145/121132.121138

[9] Burra Gopal and Udi Manber. 1999. Integrating content-based access mechanisms with hierarchical File systems. *In OSDI, Vol. 99. 265-278.*

[10] Mallik Mahalingam, Chunqiang Tang, and Zhichen Xu. 2003. Towards a semantic, deep archival File system. In Distributed Computing Systems, 2003. *FTDCS 2003. Proceedings. The Ninth IEEE Workshop on Future Trends of. IEEE, 115-121.*

[11] Microsoft. 2008. Windows Desktop Search. (2008).

[12] Aditya Rajgarhia and Ashish Gehani. 2010. Performance and Extension of User Space File Systems. *In Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10).* ACM, New York, NY, USA, 206-213. http://dx.doi.org/10.1145/1774088.1774130

[13] Margo Seltzer and Nicholas Murphy. 2009. Hierarchical File Systems Are Dead. *In Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS'09).* USENIX Association, Berkeley, CA, USA, 1-1.

[14] Lei Xu, Ziling Huang, Hong Jiang, Lei Tian, and David Swanson. 2014. VSFS: a searchable distributed File system. *In Parallel Data Storage Workshop (PDSW), 2014 9th. IEEE, 25-30.*

[15] Holt Zaugg, Richard E West, Isaku Tateishi, and Daniel L Randall. 2011. Mendeley: Creating communities of scholarly inquiry through research collaboration. *TechTrends 55, 1 (2011), 32-36.*