Implementation of a Tag-Based File System

Catherine Olsson

catherio@mit.edu
TA: Butler Lampson
11:00 am section

March 17, 2011

Introduction

This document describes TFS, a tag-based file system in which files are identified by free-form tags rather than path names or directories. In TFS, files can be retrieved from any device by their tags; additionally, files can be bundled into ephemeral, dynamic, tag-based collections called "scopes". This document describes the behavior and implementation of TFS, and provides an analysis of how TFS would perform under a variety of sample situations.

TFS has three main components: A Unix-like file representation, a hash-table-based tag lookup sector, and in-memory scopes represented descriptionally.

Major design decisions include the following:

- The basics of file storage are borrowed from the UNIX file system for simplicity.
- Efficient bidirectional lookup between tags and files is implemented by storing tag information in two places: once on the file itself, and again in the "tag sector". This approach requires additional space in exchange for faster lookups.
- A scope is dynamically-updating view, rather than a snapshot, to provide applications with the most recent data. To this end, scopes are represented descriptionally in memory, and are instantiated afresh on each request to list the contents. This approach requires additional computation time and relies on good disk caching for acceptable performance.
- By making the scopeID of a scope identical to the location in memory of the corresponding scope object, the need for a scope lookup table was averted.

Design

The main components of TFS are the files and the tag sector on disk, and the scopes in memory.

The file representation is fundamentally UNIX-like. A fixed-length *fnode* per file holds pointers to the file's data and tags. A *tag* is a free-form string. The *tag sector* is a disk sector which provides quick lookup of files by tag.

Scopes are views over files. Scopes can either be *device scopes* which hold all files on a device, or *derivative scopes* which are defined in terms of other scopes. Derivative scopes are represented descriptionally and instantiated on demand.

Disk layout

As shown in Figure 1, the on-disk layout of TFS is very similar to the layout of the Unix file system. The fnode table is a fixed-length zone holding fixed-size fnodes. The tag sector is a fixed-size hash table.

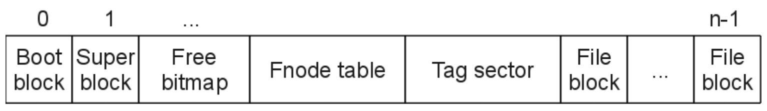


Figure 1: The on-disk layout. As compared to the UNIX file system, the tag sector has been added, and inodes have been slightly modified to become fnodes.

File representation

Files are identified on a device with a 64-bit fileID, which (like a UNIX inode number) can be used to directly retrieve the file's metadata. Files are identified across devices using the combination of fileID and deviceID; this locally-unique identifier is called a *fileUID*.

As previously mentioned, fnodes are nearly identical to UNIX inodes. However, they differ in that almost all metadata is stored as tags, to enable searching. The convention for a metadata tag is "!" - for example, "!user:Bob" identifies Bob's files. Size and number of tags are kept in fnode metadata fields, as these values are necessary to enable retrieval of data blocks and should therefore be kept close at hand.

The type of a file can be either ”regular” or ”filelist”. Filelists will be explained below, in the Tag Representation section.

Fnodes retain the block pointer structure from UNIX inodes, including indirect blocks. Additionally, fnodes contain pointers to the blocks containing the file’s *taglist*. The file’s tags are kept in alphabetical order to enable efficient search. Figure 2 shows the overall layout of an fnode and its components.

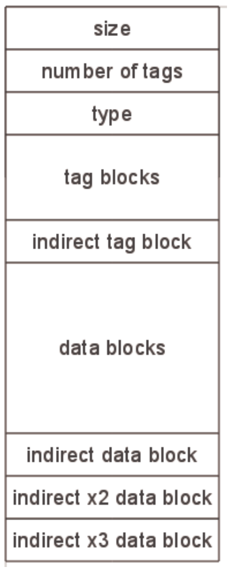


Figure 2: The contents of an fnode. Note the addition of taglist blocks, and the absence of much of the metadata, which is instead kept in tags for searchability.

The size of an fnode constrains the maximum number of tags a file can have, because the taglist has a limited number of blocks. The default fnode size allows for 3 direct blocks and 1 indirect block of file tags. If tags are stored in a 64-char field and blocks are 1KB, this allows for about 36 tags close-at-hand and 2048 tags an additional disk seek away.

The taglist is kept outside the fnodes in order to keep fnodes small and fixed-size. If every fnode were large enough to hold the maximum expected number of tags, fetching a single fnode would require fetching many blocks instead of a fraction of a block.

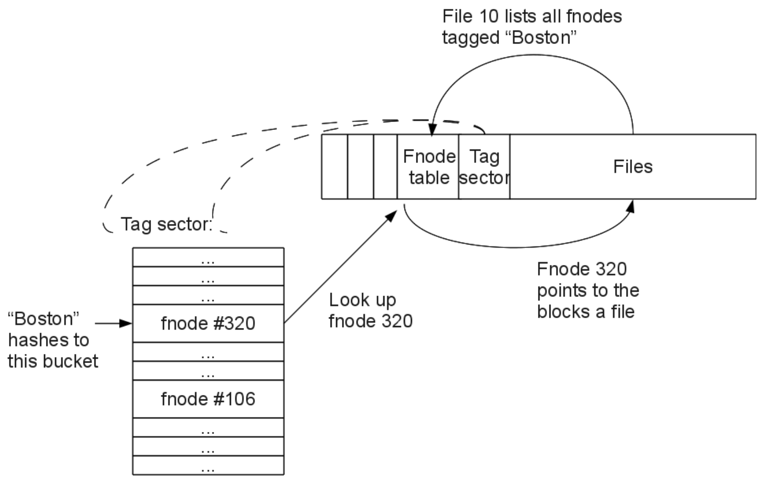
Tag representation

Tags are free-form strings. Files can have thousands of tags - they are restricted in length and number only by the constraints of the taglist.

File lookup by tag is supported by the tag sector, a hash table mapping tags to a filelist-type file containing an ordered list of fileIDs with those tags (the *filelist*). A filelist-type file is the same as a regular file, except it is not tagged and the user cannot view it or access it in any way.

The use of a hash table makes it possible to efficiently access the tags for a file. This retrieval will be efficient so long as the hash table has least twice as many slots as the number of distinct tags on the device. Figure 3 shows how the tag sector is used to map a tag to a list of files with that tag.

The filelist is itself kept in a file, rather than just a linked list of blocks, for two reasons. One, files have support for seeking in a file, which enables binary search, aiding addition and removal of tags that have very long filelists. Two, using a file means the maximum number of files per tag can be large.



File 10 lists all fnodes tagged "Boston"
Fnode table
Tag sector
Files
Tag sector:
...
fnode #320
fnode #106
"Boston" hashes to this bucket
Look up fnode 320
Fnode 320 points to the blocks a file

Figure 3: Accessing the tag sector for tag lookup. The tag sector is a hash table mapping each tag to a file containing the list of fileIDs with that tag.

File operations

The eight file-related system calls are implemented as follows:

- `read(fileUID, offset)`: As on UNIX, the fnode for the given fileUID is retrieved, the offset is converted into a block number and block-relative offset, and that block is found by traversing the direct and indirect blocks indicated by the fnode.
- `write(fileUID, offset)`: As above, but bits are written instead of read.
- `create(deviceID) -> fileID`: As on UNIX, an fnode is created and returned. Additionally, basic system metadata tags (such as `!user:Bob`)

are added. See `tag_add`.

- `delete(fileUID)`: All tags are removed from the file and its blocks are marked as free. See `tag_remove`.
- `tag_add(fileUID, tag)`: The tag is added to the file's taglist, and the tag is looked up in the tag sector and added to that tag's filelist.
- `tag_remove(fileUID, tag)`: The tag is removed from the file's taglist, and the tag is looked up in the tag sector and removed from that tag's filelist.
- `tag_get(fileUID)`: The file's taglist is retrieved and returned.
- `device_list()`: An in-memory array holds the device IDs for all mounted devices; its contents are returned with this system call. Devices are noticed by a device daemon and added to the list as they are plugged in.

It's worth noting that many operations, such as `tag_add`, are not atomic. Therefore, an interrupted read or write could leave the file system inconsistent, requiring a fsck.

Scope representation

Scopes are an emphemeral, dynamic way to define groups of files based on their tags. Each scope is represented in memory by a scope object. The *scopeID* of a scope is identical to the address in memory of its scope object. Scopes are represented in memory only by their description, and are not instantiated until the `list` system call is invoked.

Derivative scopes are built from other scopes using repeated `search` calls. Each `search` call adds a new criterion to the destination scope: for example `search(0x3333, [‘‘Boston’’, ‘‘Cambridge’’], 0x1234)` updates scope 0x1234 to include all files in scope 0x3333 matching “Boston” and “Cambridge” in addition to whatever scope 0x1234 included previously.

Note that each criterion is an `AND` of tags that must match (eg, “Boston” `AND` “Cambridge”) and the overall scope represents an `OR` of criteria that can match (eg, “Boston” `AND` “Cambridge” from scope 0x3333, `OR` “2007” from scope 0x2000, `OR` “2008” from 0x2000). Therefore, scopes can be represented in memory using an expandable list of criteria, each of which has a set list of tags.

Device scopes are a special case. Rather than having a list of criteria that defines their membership, their membership is defined by the device contents. Therefore, the kernel must check the “isDevice” flag on each scope to determine whether to treat it like a derivative scope or a device scope. For a TFS drive, the tag sector holds the underlying tag-to-file mappings. For a non-TFS drive, each fully qualified file name is considered a tag, enabling similarly efficient tag-to-file matching.

Figure 4 shows the pseudocode for a scope object in memory based on this schema, and Figure 5 provides a pictoral view.

```
struct scope {
    boolean isDevice;
    List<scope_criterion> criteria;
}

struct scope_criterion {
    scope_ID source;
    string tags[];
}
```

Figure 4: Scopes are relatively simple objects in memory. The isDevice flag distinguishes device scopes from derivative scopes. Derivative scopes have a list of criteria, *any* of which can match. Each criterion has a source scope ID and a list of tags, *all* of which must match.

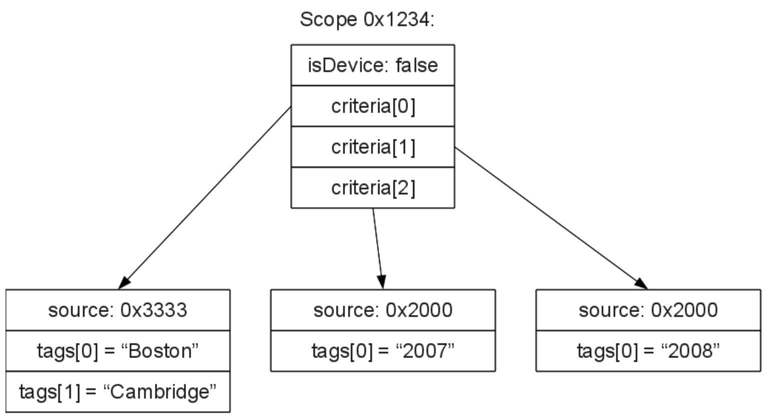


Figure 5: The structure of a scope

Scope operations

The four scope-related system calls are implemented as follows:

- `mkscope() -> scope`: Creates a new scope, by allocating a new scope object in memory and returns its memory address.
- `search(source, tags[], dest)`: Adds files to a scope, by adding a new criterion with the given source scopeID and tags to the specified destination scope object.
- `scope_tags_get(source, tag[])`: Returns a list of the fileUIDs from the specified source that are associated with all the given tags.

If `source` is a TFS device scope, each tag is looked up in the tag sector, the resulting fileID lists are merged by finding their intersection, and then the deviceID is added to each fileID of the merged list to produce a list of fileUIDs.

If `source` is a non-TFS device scope, each fully qualified filename is a tag, so only one-tag searches will yield any results.

If `source` is a derived scope, a recursive `scope_tag_get` is performed for the given tags along with all the criteria tags on each of its sources; second, the returned lists are merged together to find the intersection. A `scope_tag_get` with no tag arguments is equivalent to a `list` call.

- `list(scope)`: Returns the fileUIDs for all files in a scope. For a device scope, this call just walks the entire device and returns all the files. For a derivative scope, this call performs a `scope_tags_get(source, tag[])` call for each criterion, and merges the results to remove duplicates.

Analysis

Scenario-specific analysis

The following examples illustrate the behavior of this system in particular use cases.

Photo viewer example

For a user to view photos from a USB drive in his photo viewer, the viewer must invoke `search(USB_device_scope, [``photo''], viewer_scope)`. If the user instead wants to view all his photos tagged "Colorado" and "2007" across all devices, the viewer must invoke `device_list()`, and then perform a search on each to bring the photos into view: `search(device_scope_i, [``photo'', ``Colorado'', ``2007''], viewer_scope)`. When `list(viewer_scope)` is called, the selected photos will be retrieved and can be displayed.

Read-only media

To view all the contents of a read-only CD in a photo viewer, the viewer can perform a zero-tag search command: `search(CD_device_scope, [], viewer_scope)`. When `list(viewer_scope)` is called, all the files on the CD will be listed along with whatever else was being viewed.

Shell commands

To run a command, the shell will look in the "path": a list of (deviceID, tag[]) tuples indicating scopes that should be searched for a file tagged with the command name. A convention will be used in which the `binary` tag will indicate an executable. If there are multiple files found, the most recently created one will be run. To run a command from a different location, the shell will offer an interface for temporarily modifying the path.

Python example

The python interpreter will likewise have a path along which it will search for its modules. If python attempts to load a module and finds multiple options, it can compare their tags that start with “!ctime” to find the most recent one.

Python can also adopt other conventions for its path. For example, it could choose to use its version string as an additional tag in its path to avoid confusing its modules with modules created by other versions of python.

Broad performance analysis

Space

Space-wise, the disk overhead of TFS is comparable to that of the UNIX file system. fnodes are comparable in size to inodes. Additionally, the taglist for each file will be a compact representation of the tags on the file, storing the tags one after another in 64-byte slots, taking up no more than a few additional 1K blocks for an average file with fewer than 32 tags.

The tag sector is also compact. Assuming a 100 GB hard disk, there should be no more than 1 million files, and no more than 4 times as many tags as files. Under these assumpions, a hash table of twice as many slots as tags - about 8 million - with linear probing should be acceptable. Each slot would store an 8-byte pointer to a file containing the tag and its filelist. This would make the entire table take up about 8 bytes * 8 million = 61 MB in the worst case.

Time

First of all, it bears noting that the entire tag sector can fit in memory; additionally, frequently sought filelists are also likely to be cached. Therefore, looking up common filelists is fast, whereas less common filelists should take about one disk seek time - 10ms.

Performance analysis for most of the file operations - such as `read(fileUID, offset)` and `tag_add(fileUID, tag)` - is relatively straightforward. However, it bears noting that deleting a file will take approximately as many disk seeks as the file has tags. Therefore, a performance bottleneck will occur around deleting files with vast numbers of rare tags.

The largest area of performance concern is in the lazy construction of a list of files from a scope representation. When `list` is invoked, the scope will recursively call `scope_tags_get` on its source scopes, accumulating tags that must be retrieved, until bottoming out at the device level. At the device level, the number of accumulated tags that must be retrieved is exactly the number of tags that feature in the scope's definition. Once the filelists have been retrieved from disk, all the merge operations are performed in memory. Because filelists are stored in fileID order, these merge operations are each $O(n)$ in the length of the lists being merged, which will certainly not take a comparable amount of time to the disk operations. Therefore, a `list` operation will take about as many disk operations as the number of tags that factor into the scope defintion. If scope definitions are much more complicated than 100-1000 tags, then the `list` will not scale, and a new representation will be needed.

Conclusion

In summary, TFS offers compact and efficient support for tag-based operations, and can support anticipated workflows such as viewing photos or running python. Future directions for this work include expanding the interoperability with non-TFS systems and improving the performance of scope instantiation with a specialized cache.

Additional considerations that must be handled include freeing scopes once they are allocated, security, and hardware failures. Otherwise, this system is ready to be implemented.

Acknowledgements

Thanks to Zev for proofreading my paper, and to Syed for suggesting a CNF-like representation for scopes and noticing that merging ordered lists is relatively efficient.

Word count: 2676, sorry!