Cameleonica: Conceptual Design
Functional and quality requirements for acceptable usability

Arkadiusz Bulski

Abstract

This paper describes a conceptual design of the project, that is the functionality, the semantics, the characteristics, and the quality that will be offered to end users. Requirements are presented and categorized by features. Rationale for some fundamental decisions is explained. Operational scenarios are presented, followed with acceptance tests to establish an acceptable level of usability.

Introduction

Cameleonica project is an innovative filesystem design. Both new commands and well established commands with new semantics are given into hands of end users. Both are highly specialized in implementation. Consider the following:

Commiting and reverting history of changes is typical of version control systems, with Git as example. Encrypting entire disks is quite common, with TrueCrypt. Hiding files and even entire systems on disk is possible, again with TrueCrypt. Compressing and hashing files is a standard operation, with built-in utilities. The list of features is long.

Features are often available separately. Both filesystems and user-space utilities usually implement only one of the features at a time. User may want to use all of them. He would then need to build a composition out of several products. This is not a good way to go about it for two reasons. One is, this is hard work and end user probably expects it to work out-of-the-box. Second is, end user is not in a position to make the whole thing work optimally. Optimal system is not equivalent to a system made of optimal parts, the second is what the end user gets in this case.

Furthermore, features are often implemented outside of the filesystem. This approach is good for simplicity, as underlying filesystem needs to implement only very basic set of operations. However, an opportunity is missed this way. Filesystem is put into a position where it can provide guarantees that may not be achievable to unprivileged programs. This is because filesystem has access to the underlying storage device, and also it can precompute and cache data for more than one application in a trusted manner. This project aims to offer exactly that.

Whole project can be defined in few (somehow separate) parts: backend system calls and language bindings, semantics and features, frontend command-line and graphical tools with which users can manage entire filesystem, on-disk format to which other implementations can refer to, standards compliance, etc.

System call interface

Processes gain access to files through a long existed mechanism called syscalls. For any filesystem to be accessible, certain syscalls need to be implemented. POSIX and further standards specify the details. We shall skip the specifics of what exactly needs to be implemented and how, and generally say that following syscalls will need to be provided:

> *open, creat, read, write, pread, pwrite, readv, writev, lseek, dup, close, chown, chmod, fallocate, ftruncate, utime, link, symlink, unlink, mkdir, rmdir, rename, fsync, fdatasync, fsyncdir, sync, access, stat, statfs, fcntl, ioctl, flock, getdents, listxattr, getxattr, setxattr, removexattr, quotactl, getrlimit, setrlimit*

List above may not be accurate. Some syscalls may be removed or added later on. The list only provides a basis for initial implementation. Further, more elaborate evaluation of what needs to be provided to meet expectations of super users and comply with POSIX related standards will be needed. Other, modern, popular filesystems may be used to define how these syscalls should work, in addition to on-paper specifications.

Above that, syscalls may be subject to semantics related to specialized features that are Cameleonica-specific and have no counterpart in other filesystems. How operations are affected by these semantics will be defined in further sections, with regard to filesystem as a whole rather than single operations.

Command-line interface

Syscalls can be used to access files within a mounted filesystem but the filesystem must first be created with command-line utilities. Further long-term management of the filesystem would also be done using utilities. Following operations need to be available:

- ✔ initializing a filesystem within a container file (regular file or block device)
- ✔ changing volume settings offline (masterkeys)
- ✔ mounting a possibly interrupted filesystem
- ✔ querying file details (fragment mappings)
- ✔ changing file settings online (auto-compression)
- ✔ checking integrity, defragmenting, compressing files
- ✔ management of snapshots and revisions
- ✔ serialization into a compact incremental backup
- ✔ resizing and moving underlying partition on-line
- ✔ emergency recovery of a broken filesystem
- ✔ destruction of a decommissioned filesystem

All operations must be accessible though command-line, either through basic system utilities (cat, dd) or with specialized commands listed above.

Specific details of what information should be presented to user and what changes are allowed will be defined in further sections, when discussing specific features.

Graphical user interface

Most management operations are traditionally done with command-line tools. There are however few applications where information may be presented more intuitively with graphics. Following operations could also be available through graphical tools:

- ✔ configuring per-volume and per-file settings
- ✔ checking integrity, defragmenting, compressing files
- ✔ management of snapshots and revisions

Specific details of what information should be presented to user and what changes are allowed will be defined in further sections, when discussing specific features.

We shall consider graphical tools as convenient extension of command-line tools. Operations must primarily be accessible through command-line.

Programming language interface

While management can be done with command-line tools, sometimes operations need to be performed in a more automated way. Program sources usually do not refer directly to syscalls or commands, but rather to language-specific modules that act as wrappers. Cameleonica-specific features should be accessible in the following languages, the more the better:

- ✔ C, C#, Java, Python, Ruby

Operations on both individual files and filesystem as a whole should be available via bindings, preferably in a form of static methods. Most important operations would include at least the following:

- ✔ querying file details (fragment mappings)
- ✔ changing file settings online (safety semantics)
- ✔ checking integrity, defragmenting, compressing files
- ✔ management of snapshots and revisions

End user needs assumptions