Using Tcl Virtual Filesystems for Complex Data Management

Stephen Huntley
stephen.huntley@alum.mit.edu

Introduction

Virtual filesystems have appeared in various forms on a range of operating systems over the decades; their appeal is easy to understand – anyone who has used an FTP client can envision how nice it would be to use ones favorite file manager to move files to/from the FTP site instead of a separate program. But in a world where the savvy hacker demands complete control of almost every other aspect of the computer environment, virtual filesystems have never taken hold in widespread fashion, and computer users are largely helpless to affect how their files are stored and retrieved once they pass the bourn of the filesystem.

The TclVFS package offers a new opportunity to exploit the potential power of the virtual filesystem concept. The TclVFS package is a good fit with the mindset of the Tcl enthusiast because like Tcl in general, a Tcl virtual filesystem can act as an excellent "glue" that ties together disparate systems using simple abstractions. In addition, working with the TclVFS package has allowed exploration of powerful new techniques for utilizing the virtual filesystem concept which will be of interest to programmers who understand and appreciate the so-called "Unix way" of approaching computer problems; that is (in brief), dividing tasks into small modular units of code that work independently and do one thing well with clear simple interfaces for passing data from one module to another. In practice, making use of Tcl virtual filesystems has allowed me to solve otherwise intractable computer problems.

The TclVFS package

The TclVFS package is delivered as a stubs-enabled binary library coded in the C language. It compiles and is available on many platforms. It is an integral part of the starkit package – it allows an embedded metakit database to appear as a filesystem to the Tcl code included in the starkit. The starkit package is probably the best current demonstration of the potential for virtual filesystems to enable innovative solutions to computer engineering problems.

When the TclVFS library file is loaded, the installed TclVFS code intercepts all calls made to native Tcl C-level functions which access the computer operating system's native filesystem. Thus all Tcl commands which operate on files and directories (file, glob, cd, pwd, etc.) are dependent on the TclVFS code for fulfillment. By default the TclVFS code does nothing but pass calls to native filesystem-accessing functions through without alteration.

By providing a "mount" command, TclVFS allows a Tcl programmer to define a new branch of

the file space hierarchy visible to the Tcl interpreter, and instruct all intercepted function calls made into that file space to be redirected to a "handler" procedure written and specified by the programmer. TclVFS expects the handler procedure to conform to a defined API which takes the form of a series of handler subcommands each with its own set of arguments and expected return data format.

Format of TclVFS mount command:
VFS::filesystem mount ?-volume? path command
-volume: mount VFS as new virtual volume *path: location in existing file space of new virtual file space* *command: name of handler procedure*

Creating a new TclVFS virtual filesystem consists of writing an appropriate handler procedure which fully instantiates the defined TclVFS API. The TclVFS code ensures that all function calls intercepted by it can be fulfilled by the results of a call to one or more of the handler subcommands. Once a new virtual filesystem is brought into existence by the programmer using the "mount" command, the TclVFS package decides if a function call requires access to the native filesystem or to the virtual file space, and redirects the function call as appropriate.

Definition of TclVFS handler API:
vfshandler access *root relative actualpath mode* vfshandler createdirectory *root relative actualpath* vfshandler deletefile *root relative actualpath* vfshandler fileattributes *root relative actualpath ?index? ?value?* vfshandler matchindirectory *root relative actualpath pattern types* vfshandler open *root relative actualpath mode permissions* vfshandler removedirectory *root relative actualpath recursive* vfshandler stat *root relative actualpath* vfshandler utime *root relative actualpath atime mtime*

It is important to note that virtual filesystems mounted via the TclVFS package are visible only to Tcl code running in the Tcl interpreter in which the mount command has been executed. An optimistic but wrong assumption has been made a number of times by prospective users that a TclVFS filesystem is visible to the entire operating system and available to all compiled programs a user might want to run.

Challenges of existing Tcl virtual filesystems

The example virtual filesystems included in the TclVFS package illustrate the breadth of its ability to act as glue and tie together widely disparate technologies into a single familiar interface. The included examples are:

- a FTP file transfer VFS

- an HTTP file transfer VFS
- a metakit database VFS
- a tar archive VFS
- a Tcl namespace VFS
- a WebDAV file transfer VFS
- a zip archive VFS

In theory, each of these virtual filesystems could be mounted at the same time each in its own location, and files and directories moved and copied freely amongst them.

On closer inspection, the examples only partially implement the handler API (each in its own way), with some subcommands left incomplete and others returning only canned, static values. Most are implemented as read-only filesystems, with any attempted write action simply returning an error. Each VFS is useful as is for certain limited tasks, but if one wants to experience the full benefit of the filesystem abstraction and unrestricted information interchange with these disparate protocols, much work is left for the individual programmer to do.

The goal I hoped to accomplish using Tcl virtual filesystems was to enable complex file management, storage, archiving and transmission tasks; of the sort heretofore handled piecemeal and incompletely by programs such as CVS, rsync and tar. I knew accomplishing my goal would require aggressive combining, stacking and grafting of VFS trees into and onto one another, and I knew that for these tasks I would need the full theoretical power and function of the TclVFS API. Therefore I began my investigation of the TclVFS package by trying to establish whether that full power was indeed available.

A template virtual filesystem

Much of the value of the virtual filesystem concept comes from the fact that almost every computer user from hobbyist to professional is familiar with the abstraction it constructs: a hierarchy of files and directories. Most computer users are familiar with reading, writing, moving and copying files. Thus virtual filesystems enjoy an unusually low barrier of familiarity to adoption and use.

But in practice, incomplete implementations keep the VFS implementation barrier high. A user who innocently tries a very familiar task like copying a file faces frustration if the target virtual files space does not (for example) support write functions. Freedom to play and experiment with new combinations suggested by the VFS concept is a desirable objective.

Bearing this in mind, as well as the fact that despite its wide adoption and heavy use as a component of the starkit package TclVFS is still officially in the beta stage, one of the first goals I set for myself in the process of learning to use it was to see if the API could indeed be implemented completely before investing significant effort in trying to develop new functionalities. The idea was to produce a “hello world” sort of VFS, a completely if trivially functional system.

The first draft of the template virtual filesystem performs a very simple but nevertheless potentially useful task: when mounted it defines a virtual file space location simply as a mirror of

a real file space location. All information requested by the TclVFS package about a particular file or directory in the virtual space (by a call to the handler procedure) is fulfilled by a parallel query to the analogous file in the real space (using the standard Tcl file access commands), thus functioning in effect in a manner similar to a "symbolic link."

Initial results and conclusions

Writing the template virtual filesystem was a useful exercise. I believe it was the first virtual filesystem written for the TclVFS package to implement a handler procedure completely, with live data returned for all calls. The results were not only helpful in later VFS development efforts, but were an instructive illustration of the problems typically found in new software tool APIs. The problems found fell into three classes:

- malfunctions (bugs)
- functions that worked as stated but did not provide necessary solutions (misfeatures)
- degradation of performance when use is scaled up in new (but theoretically permissible) configurations

The problems considered in greater detail:

Bugs:

One useful result of writing the template VFS was the discovery of several bugs. The good news was that, for a beta product, the number of bugs was relatively small. The bad news was that some of those bugs were significant. One error condition resulted in a crash and termination of the tclsh shell (this bug has been fixed by the TclVFS developer team).

Perhaps the worst as-yet unfixed bug is the fact that file channels opened for writing (i.e., modes **w** and **a**) cannot be read from in the procedure called when the channel is closed. The TclVFS documentation explicitly states that seek and read operations can be performed on channels passed to the close procedure, and indeed that the ability to do so (quoting from the TclVFS docs:) "is not only crucial to the cleanup of any resources associated with an opened file, but also for the ability to implement a filesystem which can be written to." However, the necessary adjustments for reading from a write-enabled channel are not done, and attempting read operations on such a channel in the close procedure results in an error.

I worked around this issue in the template VFS by silently changing all requests to open a file in **w** or **a** mode so that the file was actually opened in mode **w+** or **a+**, respectively. These modes permit both reading and writing operations.

Misfeatures:

A misfeature is a behavior that works as designed and described, but fails to solve a user's problems when put to practical use. Trying to develop a fully write-enabled virtual filesystem revealed a potentially catastrophic misfeature:

The reader may notice that among the subcommands of the handler procedure API, no command is defined to handle the closing of a channel. The "open" subcommand is expected to pass back as part of its return value the name of a procedure to be called when the channel is closed. This "close" procedure is executed as an idle callback, and thus any errors that occur are background errors and are not visible to the user or developer.

Since the close callback procedure is often the crucial place where information committed to the channel by the user is processed in the unique ways that make a virtual filesystem useful, a silent failure can result in undetected loss of data. Any software system that permits undetectable data losses cannot be relied upon for serious work.

I tried to deal with this issue in the template VFS in a somewhat unsatisfactory manner by trying to foreground the callback errors using special error info variables and command traces.

Performance and scaling:

The advantage of the TclVFS API is that it distills all the various possible complexities of filesystem interactions in the use of Tcl's file access commands into a few simple bite-sized functions that can be handled individually and called over and over in any sequence in order to fulfill any file operation's needs.

But when a single command at the user level results in many calls to the TclVFS handler, and when one VFS happens to access a file space for information which is itself a VFS, then several such “stacked” virtual filesystems can suffer significant performance degradations.

The most egregious example of such degradation is in the handling of file attributes. A Tcl user can retrieve one or all a file's attributes with a single command. But on the level of the TclVFS package interface to the handler procedure, requests to retrieve file attributes are fulfilled by making a call to the handler procedure to discover all attribute names, then a separate call to retrieve each attribute value. Therefore for a user to retrieve all the attributes of a file in a VFS on a Unix platform, one command is mapped to four handler calls. On Windows the number of handler calls required is seven.

This multiplication of procedure calls is hardly noticeable, but if one virtual filesystem is stacked on top of another, to resolve a user's request fully requires seven times seven calls – and if one wants to stack multiple filesystems atop one another, the number of required calls explodes geometrically. Performance degradation can quickly become unacceptable.

Since I was strongly interested in stacking virtual filesystems to achieve complex file management goals, I provisionally dealt with the performance issue by setting up a cache variable to store file attributes values, so that a request for seven attribute values by a user requires no more than seven handler calls per VFS layer, keeping performance scaling issues manageable.

Applications of the template VFS

The template virtual filesystem evolved from an investigative attempt to create a fully functional TclVFS API handler into a tool to isolate and hand-hold the beta quirks of the package in order to make it as easy as possible to write new virtual filesystems. Once the most difficult problems were dealt with I developed a number of new virtual filesystems using the template VFS as a starting point:

- a collating VFS – similar to the Unix UnionFS, this VFS merges the contents of two or more disparate directory locations and makes them all appear as one directory. This VFS is designed to allow independent selection of read locations and write locations, to give fine control to acquisition and distribution of files.

- an ssh VFS – similar to the FTP and WebDAV virtual filesystems, but file information is transported via an ssh shell. This is not a pure-Tcl VFS, a ssh client binary program is required for actual communication with an ssh server on a remote computer. But all commands executed on the remote server require only standard GNU file utilities, so in general no installation or preparation is required on the remote ssh server to turn it into a secure network file server.
- a quota-enforcing VFS – similar to the quota function available on the Linux OS; but where the Linux quota only allow setting limits on total file size and inodes, this VFS enables much more fine-grained control. Quotas can be set on any file attribute reported by the "file attributes" or "file stat" commands, such as size, mtime, group id, etc.; as well as file name. Application of quotas can be restricted to certain files whose names or attributes fit specified string match patterns, alternately rules can be written to apply standards of any complexity in deciding whether a file fits under a quota.
- a "chroot" VFS – this VFS takes advantage of the facts that a virtual filesystem can be mounted as a virtual volume, and that a virtual location can be mounted on top of a real location. Mounting this VFS approximates a Unix "chroot" command, which makes a program think that a subdirectory of the root is the root. When the virtual location is specified as "/" and the "-volume" command line switch is used, then "/" is treated as a virtual directory and Tcl code running in the interpreter can only get information about the files therein from the handler procedure. The handler procedure itself though is Tcl code, so it has to perform the stupid VFS trick of unmounting the VFS on the fly every time it is called, collecting the needed information to fulfill the subcommand request from the subdirectory specified at mount time, then remounting the VFS before returning.
- a versioning VFS – This VFS keeps a unique timestamped copy of every version of a file saved to the virtual space. The handler code controls visibility of all the duplicates of the file and by default only allows the latest one to be seen. The entire VFS can be made to appear as it existed at a former point in time by setting the value of a timestamp variable. Identifying tags can be added to individual edits as well, analogous to how CVS applies tags to file versions.
- a delta-generating VFS – This VFS is designed to work only in concert with the versioning VFS, and is a good illustration of how virtual filesystems can be used to solve complex information management problems. In order to save disk space, I didn't want to save complete copies of every version of files saved in the versioning VFS. I wanted to generate and store binary deltas of the versions, which could be reconstituted back into the complete files on demand. When the delta VFS is mounted first, and the versioning VFS mounted on top of it, the delta VFS handler automatically detects when a new file version has been saved into the virtual space; it polls the directory for existing versions and replaces the last saved complete version with a delta generated with respect to the new arrival.

  All this is done without any interaction or participation by the versioning VFS handler. The versioning and delta-generating functions are completely segregated, and the delta VFS was developed without having to change a line of the versioning VFS. This demonstrates the value and utility of a simple data transfer interface and clean separation of function between information-handling modules.

Template VFS: the second draft

After writing several virtual filesystems, I noticed that I only needed to use a few simple Tcl file access commands with limited specified options to gather all the information needed by the TclVFS handler procedure, and that it was easier for me to envision writing a new VFS in terms of overloading those few commands than rewriting each of the subcommands of the handler.

I had started writing the template VFS with the idea of using it as a skeleton for the writing of new and useful virtual filesystems, but the need to catch and handle the bugs and misfeatures of the TclVFS package caused the template to become a mix of skeleton code and problem-handling code. I decided to separate out these two functions, and create a new version of the template VFS that would go beyond the original goal of fleshing out the handler API.

The goal of the second draft was to improve the problem handling code, and strictly segregate it from the skeleton code; the skeleton code itself I undertook to rework into a drastically simplified developer's API which consisted of overloads of simple Tcl file commands. The new simplified API would be easier for a new developer to understand and work with, because it would be almost self-documenting: the VFS writers task should no longer be to try to figure out what return values were considered acceptable by the TclVFS package; rather, the developer should simply have to provide code that duplicates the return format of the familiar Tcl commands.

The problem-handling code was expanded and refined. The close callback error foregrounding function was streamlined and generalized to handle more potential error conditions. The caching of file attributes was fleshed out into a general set of information caching routines which could be used to cache any desired file information. This comes in useful for example when running network filesystems such as the ssh VFS described above; caching file info for a limited time reduces the number of repetitive handler calls which have to be fulfilled by transmitting remote procedure calls over a slow network.

The simplified developer's API was segregated out to the point that a developer interested in writing a new VFS should never have to look at or deal with the problem-handling code, and all the code that a developer need concern oneself with can be easily listed in the table below:

Complete simplified API and code of template VFS draft 2:

```
proc close_ {channel} {return}
proc file_atime {file time} {file atime $file $time}
proc file_mtime {file time} {file mtime $file $time}
proc file_attributes {file {attribute {}} args} {eval file attributes \$file $attribute $args}
proc file_delete {file} {file delete -force -- $file}
proc file_executable {file} {file executable $file}
proc file_exists {file} {file exists $file}
proc file_mkdir {file} {file mkdir $file}
proc file_readable {file} {file readable $file}
proc file_stat {file array} {upvar $array fs ; file stat $file fs}
proc file_writable {file} {file writable $file}
proc glob_ {directory dir nocomplain tails types typeString dashes pattern} {
        glob -directory $dir -nocomplain -tails -types $typeString -- $pattern
}
proc open_ {file mode} {open $file $mode}
```

In order to create a new Tcl virtual filesystem, one needs simply to replace the body of each of the above procedures with code that will process file information as desired, taking care that the format of the return values is the same as that of the Tcl file commands replaced.

The objective: a personal backup utility

My motivation for exploring the TclVFS package was the desire to create a simple, flexible and reliable backup program for personal use. I am consistently astonished that so little choice is available in such a critical area of function. Due to lack of attractive choices, most computer professionals are just a hard drive crash or battery fire away from losing much of their life's work. Most of the computer software firms I have worked at have had expensive backup equipment and software, which was invariably discovered to be malfunctioning when it was critically needed.

In the past I have made a number of attempts to use or adapt existing software to make a personal backup utility. All the existing technologies I tried suffered from showstopping defects in function, scaling or reliability; a commentary on the general state of software development today.

On the issues of simplicity, stability, reliability and cross-platform function, I knew that Tcl was a good choice of development tool, but I didn't have the time or inclination to write an entire backup utility from scratch. When I found out about the TclVFS package, however, it occurred to me that I could use the concept of virtual filesystems to divide the backup functions I wanted into manageable, independent chunks, thus making the development process more straightforward.

And so most of the virtual filesystems I wrote based on the template VFS were done with the intent of ultimately combining and stacking them into a configuration that would serve as the basis of an effective backup tool, one which I could use for life, on the multiple computers and

operating systems I use now and will use in the future.

Once the template VFS and the specialized virtual filesystems I developed from it were in a state that appeared suitable, I stacked them and topped them off with a single-purpose “backup VFS,” written for the purpose of mounting the other VFS's in correct order and configuring and managing them dynamically.

The result: the FILTR

The resulting combination of virtual filesystems functions is a flexible and feature-rich backup tool I call the FILTR (File Inventory for Loading, Transfer and Recovery). The “backup VFS” stacks the virtual filesystems in the following order (from top to bottom):

- the backup VFS – this is a specialized VFS whose only purpose is to coordinate and configure the function of the other virtual filesystems. At mount time it takes two arguments: a work file area, and a backup file area. The work file area is where I store and edit my files on a day-to-day basis. The backup area is where the files in the work area are selectively copied to and archived.

  The backup VFS allows configuration of itself and the other virtual filesystems by use of a virtual configuration file which it causes to appear in every subdirectory, in effect a “file user interface” (FUI?). Once the backup VFS is mounted, any Tcl-based file manager can be used to view the work area and its configuration files, and any Tcl-based file editor can be used to open and edit them.

  Via the virtual config file a template VFS can be mounted in any subdirectory of the work area, creating a “symbolic link” to a location outside the work area, thus making it possible to incorporate any directory on the computer into the work area and set it up for backup. Backup is initiated by editing a value in the config file; saving the file causes the backup VFS to examine the contents and reconfigure the underlying virtual filesystems as necessary, and initiate backup or restore if specified. Thus every subdirectory of the work area can be configured, backed up and restored independently, without having to learn a new GUI; rather tools of the sort most computer users employ every day, a file manager and a text editor, are used to work the tool.
- the collate VFS – This VFS can be configured by the virtual configuration file so that at the same time the file is backed up a copy is published to one or more alternate locations. Thus if one is working on pages for a web site, a FTP VFS could be added to the list of secondary copy locations so that the pages are automatically posted to the web site at the same time they are backed up. Each subdirectory can be independently configured with its own publish locations
- the quota VFS – This VFS allows selective backup of the files in the work area. So, for example if a C language program is being developed and compiled in a work area subdirectory, the source C code text files can be backed up while excluding the compiled object files. Again, each subdirectory can be configured independently with its own set of quotas.
- the versioning VFS – This VFS allows each edit committed to the backup area to be tagged and stored independently. Thus any file or directory in the work area can be

subsequently restored to any previous state by specifying the appropriate timestamp in the backup VFS configuration files. Files can also be given project labeling tags in the manner of CVS.

- the delta VFS – Functions automatically in concert with the version VFS to generate space-saving deltas of file versions saved, thus allowing efficient long-term use of the backup repository.

Initial experiences of using the FILTR for backup tasks show that performance is good, it is easy and pleasant enough to use regularly, it works the same on multiple platforms, and most of all it is stable and reliable. In these respects it is as far as I know unique among available backup solutions, either free or commercial.

Thus my experience leads me to conclude that the TclVFS package is a superior tool for accomplishing complex hierarchical information management tasks.

The future

The TclVFS package has the potential to be an extremely powerful problem-solving tool. The range of applications for virtual filesystems has only begun to be explored. Some possible areas of exploration are discussed below:

Transport:

Virtual filesystems that comprise various transport protocols already exist, such as FTP and WebDAV. But new transport methods are proliferating rapidly, such as p2p, BitTorrent, and RSS. Virtual filesystems can help manage the confusion of multiple distribution methods by presenting a single uniform interface to the user.

Arbitrary metadata:

A large number of programs use metadata to organize information for storage, identification and retrieval. Most uses of metadata can be classified into four general formats:

- hierarchy
- attribute-value pairs
- keywords
- links

Although widespread, virtually all metadata implementations in current use are fragmentary and limited in their range of supported values. Often a program developer recognizes the value of metadata late in the development cycle, and tries to bolt on new forms of metadata support on an *ad hoc* basis with unsatisfactory results.

Watching the development of several popular wiki tools gives plentiful illustration of the typical evolution of thought concerning metadata. The only metadata supported by the classic wiki format is links between entries. Once use of a wiki scales to higher levels, the need for more metadata is often sorely felt. Many wikis now feature keywords, or “categories,” a very few implement a hierarchy structure.

By contrast the concept of the filesystem can integrate all the major metadata modes

comfortably, and virtual filesystems can be written to store and retrieve arbitrary metadata values; such virtual filesystems could offer existing programs full metadata support in one stroke, transparently, with minimal internal re-engineering.

For example, some popular wiki programs store each article in a separate file. If the server system for such a wiki were set up so that the article files were stored in metadata-handling virtual filesystems, it would acquire the benefit of full metadata support without any changes to the wiki program itself. Simple metadata setting and retrieving functions could conceivably be added to the system with relative ease.

Multiple views into information collections:

Just as a database engineer can construct multiple views into a single database which appear as optimized independent databases of their own for specialized purposes, virtual filesystems could be used to cast a wide range of existing information collections into new arrangements and formats.

Imagine for example an archive of program distribution packages, such as starkits. Imagine a virtual filesystem that made each starkit appear to be a Red Hat .rpm package, and dynamically reformatted the starkit into a .rpm file when copied to a location outside the VFS. Imagine multiple virtual filesystems all accessing the same starkit archive, but casting the packages into Debian .deb formats, or Solaris .pkg, etc. New software packaging and distribution formats seem to be proliferating as fast as file transport methods, and meeting demands for packaging is becoming a significant drain on a developer's time. Virtual filesystems offer the potential for "package once, distribute everywhere" convenience.

As another example, imagine a single version control archive of files, and virtual filesystems which made it look like a CVS repository, a Subversion site, a Git collection and an Arch source tree. Each developer interested in the contents could access them using ones preferred client program, and the archive maintainer would only have to submit changes in one format to make them available in many.

Desktop integration:

Most operating systems have some sort of virtual filesystem scheme, though each is unique to its brand, and their complexity and heterogeneity present high barriers to installation and new VFS development. The TclVFS package has the potential to be the *lingua franca* of virtual filesystem development across platforms by means of simple bridging tricks to connect a virtual filesystem driver to a Tcl interpreter.

For example, most OS virtual filesystems have a FTP module available for making a FTP site appear to be a directory in the user's file space. Since the tcllib collection of programs contains a pure Tcl FTP server, it would be relatively simple to start a local Tcl FTP server which accessed its files from a virtual filesystem mounted in the Tcl interpreter, then aim the OS-specific VFS FTP driver to read from the local Tcl FTP server.

In this way, all a user's compiled programs, not just Tcl scripts could take advantage of virtual filesystem functions. Version controlled code check-ins and check-outs could be done via drag and drop in ones favorite file manager rather than requiring specialized GUI or command-line tools. One could run Make directly on the latest makefile in the head revision of a CVS archive mounted as a VFS, making it easy to install the most up-to-date snapshot of a program. Backup functions could be automatic and transparent to the user, thus making the crucial practice of backing up ones work more commonplace.

Conclusion

Simply stated, my experience with the TclVFS package has persuaded me that virtual filesystems are such a powerful tool for simplifying the complex challenges routinely faced by computer professionals and serious hobbyists, that the package and the concept have the potential to restore Tcl to the front rank of visibility and use among scripting languages and rapid development environments. I hope the template VFS will make it much easier to develop new virtual filesystems and serve as an aid to wider adoption and utilization of the TclVFS package.

The template VFS and several of the derived virtual filesystems discussed here will be added officially to the TclVFS package' s set of examples in the near future.

Links:

- The TclVFS package is available at: http://tclvfs.sourceforge.net
- The template virtual filesystem and the VFS's developed from it, as well as the FILTR backup program, are available at http://filtr.sourceforge.net