A Reboot of the Starpack Build Process for Tcl 8.6 and Beyond

Stephen Huntley, Güneş Koru, Yili Zhang, Urmita Banerjee, Leroy Kim, Clif Flynt

Health IT Lab at UMBC
Department of Information Systems
University of Maryland, Baltimore County
Baltimore, Maryland, USA, 21250

stephen.huntley@alum.mit.edu,
{gkoru,yili.zhang,urmita.banerjee,leroy.kim}@umbc.edu,
CLIF@cflynt.com

Abstract

We describe an initiative to simplify drastically the compilation of Starkit basekits for Unix and Windows, which excises no-longer-needed code and relies wholly on Tcl tools for build scripts. Basekit code providing once-essential features is now redundant thanks to Tcl 8.6's feature set, and is eliminated. Our changes facilitate a professional standard of security auditing and accountability of basekit contents, make it easier to upgrade separate library components, and will ease future support of new platforms. Tcllib's TEPAM package was used to create a human-friendly command line wrapper, and Ghostscript's tmake.tcl (a partial clone of standard make) was used as a flexible and configurable backend compile engine.

1. Introduction

Starkits and Starpacks remain popular among Tcl developers and their clients as a simple and effective technology for delivering and deploying Tcl/Tk applications as a single file, within which all code files, resources and libraries are stored as a directory hierarchy in the form of a virtual filesystem.

A Starkit is a single-file archive of a software project. The value of a Starkit can be leveraged by turning it into a Starpack, a single-file platform-specific executable program. In order to do that a "basekit" is required, a Tcl interpreter which includes all the support files and packages the interpreter needs in a virtual filesystem appended to the end of the interpreter executable file. The basekit and Starkit are combined into a single file as a Starpack, which bootstraps itself on program startup by locating and mounting the appended virtual filesystem as part of the startup sequence.

But obtaining an appropriate basekit has in recent years become the most challenging step in an otherwise compellingly simple process. In the past, developers who did not wish to compile their own found it easy to rely on pre-compiled basekits for a range of computer platforms, provided by trusted sources. As these original sources have by and large disappeared over the last decade, their places have been taken by volunteers who offer their own selections of basekits on personal websites. In addition, a few developers have made Herculean efforts to provide software intended to make compiling ones own basekit relatively simple. But these individual efforts face the same challenges as any software project: maintenance, fixes, upgrades, etc., and efforts by volunteers are not always enough to keep everything working properly and reliably.

The trend has thus been toward increasing difficulty and unreliability in obtaining stably-functioning and desirably-configured basekits in which a developer can have the confidence to bet ones deployment strategy.

The staff of the Health IT Lab at UMBC[1] are pursuing an ambitious strategy of rapid development and deployment of health information applications in which Tcl/Tk and Starkit/Starpack technology play a crucial role. We identified continued access to high-quality basekits as a risk in our projects. We thus reviewed current available compilation tools and techniques, and found that a case could be made for re-engineering and simplifying the compilation workflow to a bare-bones level. The goal was to ensure availability, confidence in stability and future manageability of Tcl basekit executables.

Our review also showed that even the most up-to-date currently available basekits included outdated supporting libraries, some of which we found could be upgraded, and some which could be discarded entirely due to advances in Tcl's base feature set. We thus decided to take the opportunity during this re-engineering to make the first significant changes to the internal structure of basekits perhaps since their introduction over fifteen years ago. This entails leaving behind support for Tcl versions before 8.6, but we reason that the simpler the initial product is, the easier it will be to maintain and improve going forward, and we have no need in our projects to support older versions of Tcl.

Our re-engineered compilation code project was built from scratch, and doesn't incorporate any code from existing basekit compilation tools. To facilitate a rapid prototyping and development process, we used two powerful and flexible Tcl-based tools: TEPAM (Tcl's Enhanced Procedure and Argument Manager)[2] from Tcllib; and tmake.tcl[3], a partial make clone from the Ghostscript project.

2. History

The original basekit, “Tclkit”, originated with Jean-Claude Wippler and was distributed from his business web site equi4.com[4], starting from around the turn of the Millennium. The earliest build system still existing on that site is “genkit”[5], based on a 25kB Tcl script which downloaded prepackaged code tarfiles from a directory on the equi4.com site.

The genkit project contains what looks like an earlier build script, “M.sh”[6], a brief 1 kilobyte Tcl script.

Genkit was superseded in 2006 by “Kitgen”[7], which was driven by a script, config.sh, that formatted a makefile, which guided the compilation steps for creating a Tclkit.

In 2007 Kitgen became the Kitgen Build System[8], a project including a master controlling Tcl script, kbs.tcl, and individual scripts for controlling compilation of the necessary libraries as well as a number of optional libraries to be included in the final product.

Pre-built Tclkits for multiple platforms were regularly uploaded to a code.google.com[9] project page until 2010.

In 2010 the Kitcreator project[10] was released by Roy Keene. The project comprises a hierarchy of shell scripts. The scripts not only build a basekit, they, like Kitgen, control the building of several optional Tcl loadable package libraries. They also download source code from the constituent libraries’ distribution sites, apply patches to the source code and customize environment variables for control of library configure scripts and makefiles.

Kitcreator is an ambitious project that builds basekits and optional loadable libraries for multiple platforms with multiple configuration options. It appears to be the most up-to-date and actively-maintained tool for generating basekits. The author also provides a web interface[11] for generating and downloading custom basekits for those who don’t want to install and run the project themselves, which makes obtaining a suitable basekit simple for the new or busy developer wanting to get started with the technology.

3. Motivation

Since the Health IT Lab is developing applications to handle medical data, security and auditability are high-priority concerns. It is important to be able to know and verify the contents and function of the source code of the applications.

It is also important to have stable access to executable programs and libraries on all supported platforms, which entails the necessity to be able to build repeatably all third-party products used, from source code which is stored in repositories managed in-house.

But at the same time we want to be able to maintain and enhance our third-party code on an ongoing basis by incorporating new releases, which may contain important bug fixes and new features.

We want to be able to port our code to new platforms as users of the software may demand to use in the future.

After examining available projects for building basekits, we found that Kitcreator incorporated crucial information and techniques for building basekits in a modern environment. For example, it applies patches to the Metakit library source code that allow it to build against current releases of the Tcl code base.

But Kitcreator's design – its complexity, its ability to download source code from arbitrary web locations and its methods of setting and passing compilation options via environment variables through multiple hierarchical levels of shell levels – made it difficult to have confidence in what actually was in the final basekit product. In addition, generation of basekits for all desired platforms could not be reliably repeated over time, and Kitcreator's complexity makes the prospect of diagnosing and fixing build failures daunting.

4. Tools

After examining the structure of basekits and the history of tools for generating them, it became apparent that the best way to achieve the Lab's project goals for third-party open-source software was to create a new, drastically simplified compilation workflow with the sole purpose of reliably and repeatably producing the simplest possible Tcl basekits on the supported platforms (Linux and Windows), the workflow being controlled by Tcl tools that would make it easy to understand and audit the basekits' structure, and easy to debug and fix future problems and incorporate future enhancements and platform ports.

TEPAM (part of Tcllib) is a very useful tool for rapid project development. It is primarily a feature-rich utility for declaring the syntax and constraints of arguments for procedures (including switches, named and unnamed arguments). It simplifies the tasks of initial definition and later enhancement of a program's API. It has an additional feature not seen in most other Tcl-based argument-parsing tools: a self-documenting feature – a human-friendly help text automatically generated from the argument definitions. The help text is output when a precedure is called with a "-help" flag in a way that Unix power users will find familiar. This is useful for documentation and ongoing maintainability.

TEPAM also allows definitions of procedures with sub-commands, similar to ensembles. It also allows a second tier of sub-sub-commands (analogous to the built-in Tcl commands "string is integer", "string is list", etc.). Thus a script with multiple entry points can be written, giving the developer a quick way to develop, test and add features independently before final program integration.

The Tcl script program tmake.tcl is a partial clone of the standard make utility. It is part of the Ghostscript project, licensed under the GNU Affero General Public License[12]. Like standard make, it allows quick and flexible definition of a conditional workflow to build a final product out of many constituent parts, with the same syntax. Thus a tmake.tcl makefile is a valid standard makefile. Tmake.tcl is also able to include and parse simple standard makefiles, incorporating their variable and rule definitions.

The fact that tmake.tcl is pure Tcl eliminates concerns about platform dependence and behavior of different flavors of make programs. Its reduced feature set has the advantage of keeping makefile complexity from getting out of hand. It is a valuable tool for quickly adding simple workflow capability to a Tcl-based project.

A few custom edits were made to tmake.tcl to fix bugs and add a feature: the original tmake.tcl throws an error if an "include" directive specifies a file that doesn't exist. The edited tmake.tcl behaves more consistently with standart (GNU) make – an include directive specifying a non-existent file is ignored. This turned out to be an important improvement, allowing conditional inclusion of platform-specific makefiles.

TEPAM and tmake.tcl complement each other well for rapid program development. The runtime behavior of a makefile can be powerfully and flexibly controlled by specifying targets and variable values on the command line, although the available options can be mysterious. In this project TEPAM is used in a wrapper script to specify desired outputs and configurations using named argument flags, the wrapper script then passes on the necessary custom settings to tmake.tcl to guide execution of the makefile. And since a single TEPAM-based script can be written to have multiple entry points, the recipes in the makefile rules can all be calls to a single documented Tcl script containing all the logic for building all targets in the makefile, instead of a mass of confusing platform-dependent calls to the shell (as a makefile often becomes). TEPAM's support for sub-commands makes it easy to develop and test separate recipes in isolation before using them together called from the makefile.

5. Streamlining the basekit build process

The simplest basekit using Metakit for support file storage, as built by Kitcreator, consists of Tcl built as a library, the TclVFS[13] package library and Metakit[14] built as a Tcl package library (Mk4tcl); plus a project called "kitsh" (kit shell) containing code necessary for producing a custom Tcl interpreter and the features required for a basekit to work. Kitcreator allows specification of Tcl release used, but the other code base versions are fixed: TclVFS 1.3, Metakit 2.4.9.7.

The kitsh project includes:

- zlib.c
  to provide zlib compression/decompression features, necessary pre Tcl 8.6

- rechan.c
  to provide a reflected channel feature, necessary pre Tcl 8.5

- pwb.c
  contains a bug workaround for initialization of encodings, necessary pre Tcl 8.5

- main.c
  to provide the necessary "main" function for a program

- kitInit.c
  bootstrap code to mount the Metakit virtual filesystem and initialize the interpreter

For the Health IT Lab streamlining project, a directory structure was created and populated with subdirectories containing multiple versions of Tcl, TclVFS and Metakit, plus the kitsh code from Kitcreator. A makefile was written that allowed the version of each component to be specified at runtime.

The recipes for each build component's make rule were extracted from Kitcreator build logs and simplified by trial and error. The nature of Kitcreator's build process -- multiple nested shell scripts, configure scripts and makefiles – accretes settings in environment variables and results in extremely lengthy compiler command lines with mysterious origins and values. By comparing these command lines with much older basekit build scripts (e.g., M.sh, genkit), we guessed at the really important compiler settings and eliminated the rest. We found that a basekit could be compiled successfully with very short compiler command lines.

Also by trial and error, different versions of Tcl, TclVFS and Metakit were tried by setting TEPAM command-line options and passing them via the wrapper script to tmake.tcl and the makefile. We found that an up-to-date basekit incorporating Tcl 8.6.8, TclVFS 1.4.2 and Metakit 2.4.9.8 worked perfectly.

Upgrading to a recent release of TclVFS brought particular advantages. As examination of the kitsh package shows, extra libraries for reflected channels and zlib functions are no longer needed when using Tcl 8.6, but TclVFS 1.3 still relied on the commands those libraries provided. TclVFS 1.4.2 has been improved to use the new built-in zlib and reflected channel features if they are available.

Thus, with the upgraded TclVFS library incorporated, we found we were able to eliminate the kitsh zlib and rechan libraries (as well as the pwb library) entirely from the end-product basekit. Additionally, libieee (long deprecated, yet still incorporated into every basekit heretofore) is excluded from the final basekit. These exclusions significantly streamline and simplify the build process and the operation of the end product. Only a few simple edits to kitInit.c were required so that no attempt would be made to load/initialize the non-existent libraries. This simplification will make future maintenance of the basekit project easier.

6. Enabling customization

From the start, basekits have been built using Metakit as the persistent storage basis for a virtual filesystem. The developer of the first basekits, Jean-Claude Wippler, is of course also the author of Metakit. The Kitcreator project adds the option of choosing one of two other storage backends, a zip archive, and an encrypted file archive. But Metakit is the only backend that allows file writes to the virtual filesystem as part of the operation of the basekit post-creation. This is a crucial feature for the Health IT Lab's projects, since it is necessary to be able to push updates to installations in the field already in use.

But Metakit also presents challenges and issues in the life of basekits going forward:

- uncertain future support by the author
- introduces a dependency on C++
- lack of database locking makes it prone to corruption
- writes to the DB are not immediately saved on disk, commits are written after a looping time delay, producing a time lag which might lead to loss of data

It would be nice to be able to hack the basekit code to explore other back end storage technologies. It would be especially nice to do this by hacking Tcl code instead of dealing with editing and compiling C. But the bootstrap code in the file kitInit.c is hard-coded to deal with only the three pre-determined backends.

The hard-coded startup/initialization sequence for a basekit:

- main function in main.c calls Tcl_Main with TclKit_AppInit named as the appInitProc

- Tcl_Main calls TclKit_AppInit in kitInit.c

- TclKit_AppInit calls _Tclkit_Init

- _Tclkit_Init calls _Tclkit_Generic_Init

- _Tclkit_Generic_Init calls TclSetPreInitScript with the string pointer preInitCmd which points to Tcl code formatted as a C string and embedded in kitInit.c

- preInitCmd is evaluated in Tcl_Init

- preInitCmd sources boot.tcl which defines the procedure tclInit, which replaces the default tclInit procedure in Tcl_Init during standard interpreter initialization

The difficulty of booting a basekit lies in the last step, which presents a chicken-and-egg problem. The file boot.tcl must be sourced because it contains the code for mounting the basekit's virtual filesystem. But boot.tcl is stored ***within*** the virtual filesystem.

The kitsh project gets around this problem by providing custom Tcl code (for each supported virtual filesystem) that knows the details of the filesystem persistent storage and can reach into the storage area before it is mounted (via TclVFS), retrieve the information corresponding to the contents of the boot.tcl file, and source it. The specific bootstrap Tcl code for the selected backend storage option is incorporated into preInitCmd via a preprocessor include directive at compile time.

It would be useful, for the purpose of devising new basekit backend VFS options, for a developer to be able to specify none of the default VFS options at compile time and incorporate custom VFS code.

It would also be nice to eliminate the chicken-and-egg problem. Why use preprocessor include directives to embed custom code for accessing the contents of boot.tcl in a yet-to-be mounted virtual filesystem? Why not just use the preprocessor include directive to embed the boot.tcl code directly?

These two things have now been done in the Health IT Lab praoject, simply by adding two new preprocessor include directives in the preInitCmd Tcl code in kitInit.c, in a spot in the code that is not executed unless none of the default VFS preprocesser defines have been set. The first include directive is intended to contain code to define any Tcl virtual filesystem. The include directive incorporates code from a file called customvfs.tcl. The second include directive is intended to incorporate a custom version of boot.tcl that can access the form of persistent storage the developer desires, and mount it as a virtual filesystem using the previously loaded virtual filesystem code. (Of course in practice any code can be included to drive any desired customization of the basekit intialization process.) Rules have been added in the makefile to format the custom files as C strings to make them suitable for inclusion in kitInit.c

In this way, by making it possible to hack new Tcl code and incorporate it at compile time, the initialization of the basekit can be customized in unlimited ways. This may include, as one example, creation of pure-Tcl methods of both reading from and writing to storage areas appended to the basekit executable, safely and securely. This will make it easier to move beyond Metakit as a writable storage solution, if and when that is deemed desirable.

7. Future work

Starkits and basekit-enabled Starpacks have existed for close to twenty years, but still have unexplored potential. Competing application delivery technologies using other languages and their tools tend to be unreliable, bulky, messy and/or insecure[15][16][17]. Thus Starkit/Starpack technology remains attractive.

A SQLite VFS shim file has recently been committed to the SQLite fossil repository[18] that claims to allow an SQLite database to be appended onto the end of some other file, such as an executable. The new basekit hacking capabilities described in the previous section could be used to create and initialize a basekit that uses a SQLite-based virtual filesystem as a writeable file backend.

In pursuit of maximum simplicity, support of operating system platforms besides Linux and Windows has been ignored. But the combination of TEPAM and tmake.tcl has proved useful in rapid development of modular, configurable code. The Windows basekit is built by including a Windows-specific makefile into the base (Linux) makefiles, overriding rules where necessary and setting new variable values to enable cross-compilation, conditionally controlled by the TEPAM wrapper script. It should be straightforward to use this approach to develop makefiles for new platforms, such as Macintosh.

A persistent complaint of Starkit and Starpack users is the inability of non-Tcl programs to see into a Starkit's Tcl virtual filesystem and share files and libraries, and the inability of a Starkit to load non-stubs-enabled libraries straight from the virtual filesustem. The new ability to customize the initialization of a basekit presents the possibility of including Linux FUSE (Filesystem in Userspace) code that allows mounting of a Tcl virtual filesystem as a FUSE filesystem, thus allowing the operating system to handle files in a Starkit as though they were native files. The same goal could be similarly accomplished on the Windows platform by creating a Windows filesystem letter drive linked to a local FTP server, which is started using an FTP server package from Tcllib within the basekit with a custom initialization procedure.

References:

[1] https://drkoru.us/health-it-lab.html

[2] https://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/tepam/tepam_introduction.html

[3] http://git.ghostscript.com/?p=ghostpdl.git;a=blob;f=toolbin/tmake.tcl

[4] https://equi4.com/tclkit/

[5] https://equi4.com/pub/tk/tars/genkit

[6] https://code.google.com/archive/p/tclkit/source/default/source?page=7

[7] https://equi4.com/tclkit/kitgen.html

[8] https://sourceforge.net/projects/kbskit/

[9] https://code.google.com/archive/p/tclkit/downloads

[10] http://kitcreator.rkeene.org/fossil/index

[11] http://kitcreator.rkeene.org/kitcreator

[12] https://www.ghostscript.com/license.html

[13] https://core.tcl.tk/tclvfs/index

[14] https://git.jeelabs.org/jcw/metakit

[15] https://wiki.python.org/moin/deployment

[16] https://thehftguy.com/2016/11/01/docker-in-production-an-history-of-failure/

[17] https://hackernoon.com/electron-the-bad-parts-2b710c491547

[18] https://sqlite.org/src/file/ext/misc/appendvfs.c