Beyond TclKit - Starkits, Starpacks and other *stuff

Steve Landers
Digital Smarties
steve@digital-smarties.com

1 Introduction

To those coming from a Unix background, "source" is often equated with "free", and "binary" with "commercial". The reasons for this are straightforward - Unix is cross-platform, source code is the lowest common denominator, the average user is a programmer (even if they don't know it) and each machine has a compiler.

But source code is not for everyone, and end-users are not "stripped-down developers". One can see why in the Windows and Mac desktop OS world "source" usually means "hassle" and "binary" mean "convenience".

Into this world comes Tcl/Tk - a scripting language that facilitates the development of cross-platform graphical applications. But the only deployment options are source (which provides only one side of the equation) or platform specific binary wrapping (which provides the other).

What is needed is a deployment model for Tcl/Tk applications that provides the simplicity of a single file binary download with the openness of a source distribution. Such a model shouldn't force the user to become a programmer and nor should it necessarily mean applications become closed. And it should be a deployment model flexible enough to effectively support various deployment media - from network based installation to CD-ROM installation.

This paper describes the experiences with Starkits - single file packaging of Tcl scripts, platform specific compiled code and application data - and TclKit - a single file Tcl/Tk interpreter. The benefits of this approach compared to the alternative wrapping strategies are addressed, as is the ability to build executables for multiple platforms from a single platform.

The paper shows how to construct a Starkit and shows how to construct cross-platform Starkits containing compiled extensions for several platforms. It also looks at some advanced topics such as the architecture of a Starkit, code privacy, database management, adding help to a Starkit and installation options.

But the paper goes beyond TclKit and looks at how these benefits could be realised by all Tcl/Tk applications. It suggests a few core changes that would make this feasible.

And finally, it looks at issues and benefits of a Starkit repository.

2 Background

Binary versus source, RPM vs apt, tar vs cpio. It wasn't always like this in the Unix world. Consider shar - the shell archiver - which became ubiquitous in the early days of the Usenet. A shar archive is a single file shell script containing a compressed binary archive (usually uuencoded). This revolutionised the distribution of Unix software - a single file download which, when run, could build and install an application. But shar files weren't suited to more complex applications. Packages like RPM[1] or Apt[2] help - but they are very much geared to solving a system level problem not (potentially) cross-platform application deployment. And there is still the source/binary dichotomy.

Things are not quite so bad in the Windows and Mac worlds. Being single platform helps, since the user base is far less interested in source. Of course, these platforms have their own idiosyncrasies (even when dealing with binary distributions) - hence the development of products like Vise[3] and InstallShield[4]. But there is still the potential effect on system stability when application installers manipulate the Windows registry.

Scripting is freeing people to choose the best platform for their application. Developing in Tcl/Tk can be very productive - providing high levels of functionality covering all aspects of business logic, graphical user interfaces, databases, networking and interfacing with existing technology. And, with a little care, Tcl/Tk applications can run well on Windows, all common Unix variants and the Macintosh.

Scripting delivers on the promise of making programmers more productive - and so deployment is becoming more of an issue, for both open and closed source applications.

3 Deployment

Users have come to accept that computers can be enormous time wasters - especially when it comes to installation of software products and updates. When you think about it, the whole concept of "installation" is artificial and unnatural. When was the last time you installed an appliance like a toaster or kettle? More importantly, why do you need to navigate through a labyrinth of jargon and acronyms just to try a package? And why should you care where it is installed? And when you decide you don't want it, why is it so difficult to get rid of it and its detritus?

Perhaps the answer is that deployment has been viewed from the perspective of the developer. Consider what is needed when deploying a Tcl/Tk application:

- a Tcl/Tk interpreter for the target platform
- any compiled packages for the target platform
- any Tcl packages and, finally,
- the application scripts themselves

Note that this mirrors the developer's installation environment - individual components installed into the host filesystem.

Getting these onto the target machine originally involved installing a Tcl/Tk distribution (or, in the case of popular versions of Linux - relying on one being installed). Then one would unload a tar or zip file into a known location - hardly a general end-user solution. On Windows it was a little nicer - Tcl/Tk was deployed using a proprietary installer, and the same could be done with application code. But no matter how much eye-candy one adds to it, the concept of having to set up a runtime environment in addition to the application itself can be a stumbling block for many end users.

More recently there have been a number of "wrapper" tools produced for Tcl - most notably ProWrap (part of TclPro [5]) and freeWrap[6]. These address the problem of installing a Tcl/Tk runtime by taking a Tcl/Tk interpreter, application scripts, compiled extensions and data files and producing a single file executable.

They also provide other benefits:
- since the application is self contained, it won't be broken by the installation of newer components (e.g. a more recent Tcl/Tk or extension)
- they can provide a degree of code privacy - freeWrap through (admittedly) simple encryption and ProWrap through bytecodes

Yet despite their benefits, such wrappers do have their limitations:
- they don't address multi-platform deployment scenarios
- the only update option is full replacement (of what is a rather large executable)
- wrapped applications tend to run in a different context to the development environment, and so need additional careful testing

This is not to devalue these programs - freeWrap in particular has a number of satisfied users. But in many situations it is important to go further and address the above limitations.

What is needed is a deployment method for Tcl/Tk applications that provides:
- platform independence
- supports source and binary distributions
- supports compiled extensions
- supports single file deployment or can use an installed Tcl/Tk interpreter
- supports compression to reduce distribution size
- supports a mechanism for code privacy
- can be launched without unpacking
- run in the same or similar context to the development environment
- support incremental updates as the product matures

It is this set of requirements that has driven the evolution of **Starkits** - whose purpose is to package Tcl/Tk applications as a single file for easy distribution and use.

4 Starkits

Imagine having a simple directory and file structure, where all application scripts, standard packages, compiled extensions, documentation, images and other binary data resides. That's not so hard - in fact, it is most likely common practice among many Tcl developers already.

Imagine also having a well-defined way of storing both such an application and all its support files, and the Tcl/Tk system itself. Then deployment would become a matter of picking up all the relevant pieces, shipping it to the target machine somehow, and it would run out of the box.

Suppose furthermore that the structure could be wrapped into single self-consistent files, in a space efficient compressed form, and that these file could be "executed" without unpacking, and without altering a single line in the application.

This, in a nutshell is what Starkits are all about. A Starkit is a packaging mechanism for delivering applications in a self-contained, installation-free and portable way.

Note that Starkits used to be called "Scripted Documents", and a paper on a much earlier version of these was presented at the Tcl2000 Conference [7].

4.1 Starkits overview

The name **Starkit** is an acronym for **ST**and**A**lone **R**untime.

Starkits let you "seal" a complete directory tree into a runnable form. The Tcl/Tk interpreter which can "run" such documents is called TclKit. TclKit itself is little more than a wrapped version of Tcl, Tk, IncrTcl and a few more extensions - plus all the necessary runtime library scripts.

The essence of Starkits is that they contain a "filesystem-inside-a-file". This filesystem contains a copy of all files that form an application, but in normal use it never gets unpacked at all. The TclKit runtime contains a complete Tcl/Tk system, and it too can function without ever being unpacked. The command "tclkit myappfile" is the Starkit way of "running the application", although on Unix the same effect can be achieved simply by making the Starkit executable ("chmod +x") and having TclKit available path.

Having an application in just two files has a number of benefits:
- TclKit is a generic runtime - the same file works with all Starkits
- there is one TclKit build for each platform
- if an application contains only portable scripts then the StarKit is portable

So if multi-platform deployment is an issue, one can deploy the application as a file which runs on all platforms - just by launching it with the appropriate TclKit system. The philosophy behind Starkits is that deployment is an integral part of the development process. Applications get built and extended over time, in whatever way a developer sees fit, but with a certain directory structure as the storage convention. Development can take place using any Tcl/Tk installation (including TclKit!). Tcl commands such as "package require", "source", and "load" can be used - as always.

Then, when the time comes to deploy, all one does is wrap the entire application directory tree into a Starkit. The resulting file is compressed and ready for deployment. In fact, it is runnable. The effect is that deployed applications run in nearly the same way as they do during the development process. This is achieved through the Tcl Virtual File System facility[8] - a new Tcl/Tk 8.4 feature that allows you to divert all filesystem access away from the native operating system and to something else. On start-up, the Starkit gets “mounted” so that it appears to contain a directory structure. This internal internal filesystem can be read or written like a normal files system. The VFS "mount" concept makes running from a real filesystem and running from inside a Starkit almost indistinguishable.

4.2 Starkits - a simple example

To build a Starkit you need

- TclKit for your platform - which can be obtained from the “TclKit Home Page” at[9]
- on Windows you’ll also need tclkitsh - the console mode version of TclKit
- the SDX utility - which can be obtained via the “SDX Download Page” at [10]

SDX - the **S**tarkit **D**eveloper e**X**tension is a Starkit containing a collection of scripts for creating and manipulating Starkits. You can list the available scripts by running:

```
$ sdx help
```

Firstly, we’ll need a small Tcl/Tk script that serves as our application - in this example we’ll use the ubiquitous “Hello World” script which we’ll put in a file **hello.tcl:**

```
package require Tk
pack [button .b -text “Hello World!” \
                -command bell]
```

Note that we explicitly specify “package require Tk” - this is because Tk is a dynamically loadable extension within TclKit and must be explicitly loaded if required.

Now, we use the SDX **qwrap** command to do a “quick wrap” of this script into a Starkit:

```
$ sdx qwrap hello.tcl
```

The result will be a file called **hello.kit** - the resulting Starkit. On Windows, there’ll also be a file **hello.bat** - which just invokes hello.kit using TclKit. To invoke the Starkit, on Windows just invoke “hello” or on Unix invoke “./hello.kit”.
So, what did qwrap do for us? To find out, we use another SDX feature to unwrap the script so we can examine its contents

```
$ sdx unwrap hello.kit
```

The result is a directory called **hello.vfs** which contains a copy of the contents of the Starkit virtual filesystem:

```
hello.vfs
|-- main.tcl
`-- lib
    `-- app-hello
        |-- hello.tcl
        `-- pkgIndex.tcl
```

The first thing to note is the directory structure. SDX qwrap implements the recommended Starkit convention - storing the application code as a package with the **app-** prefix. This is done for convenience and consistency - all code within the Starkit is within a package, and application code can be easily distinguished from libraries and packages.

Main.tcl is the script that is run whenever a Starkit starts. Looking at its contents, we see sdx qwrap has generated:

```
package require starkit
starkit::startup
package require app-hello
```

The first line loads the Starkit runtime package - a small (less than 100 lines) pure Tcl package that manages the Starkit start-up sequence (amongst other things, it mounts the Starkit VFS).

In the second line the **starkit::startup** procedure is called to initialise the **starkit::topdir** variable. It also adds the Starkit **lib** directory to the Tcl **auto_path** variable, thus making available any packages stored in that directory.

And finally, the “package require” causes the hello.tcl script to be sourced from within the lib/app-hello directory inside the hello.kit VFS.

Note also that qwrap has created the lib/app-hello directory and copied hello.tcl into there (adding a “package provide app-hello” line if necessary) and created a pkgIndex.tcl file to enable auto loading by Tcl. If there is no “package provide ..” line in the source code qwrap will add one to the pkgIndex.tcl for you.

We can now make changes to hello.tcl, but we have two copies - our original one, plus the copy under hello.vfs.

There are two ways to address this:

- we can either remove our original hello.tcl and just make changes to the one under hello.vfs. script.
- alternatively, a convenient technique for developers (at least, on Unix) is to replace hello.vfs/lib/app-hello/hello.tcl with a symbolic link to the hello.tcl script

Assuming we have made changes, we can recreate hello.kit using the SDX **wrap** command:

```
$ sdx wrap hello.kit
```

4.3 Using packages in Starkits

Using packages within Starkits is only slightly different, and relies on the Tcl **auto_path** variable being set for us by the **starkit::startup** script.

As an example, we'll modify our hello.tcl script to use gButtons - the fancy Tk buttons package [11] - which is itself packaged as a Starkit.

There are two components we need to get from the gButtons Starkit - the gbutton and the autoscroll libraries. So, we download gButtons, unwrap it (using "sdx unwrap") and copy both the lib/gbutton and lib/autoscroll packages to hello.vfs/lib. This gives us the new directory structure:

```
hello.vfs
|-- main.tcl
`-- lib
    |-- app-hello
    |   |-- hello.tcl
    |   `-- pkgIndex.tcl
    |-- autoscroll
    |   |-- autoscroll.tcl
    |   `-- pkgIndex.tcl
    `-- gbutton
        |-- disabled.gif
        |-- down.gif
        |-- gbutton.tcl
        |-- pkgIndex.tcl
        `-- up.gif
```

Note that the gbutton directory includes both Tcl scripts and GIF images. A Starkit can contain scripts, images, data and (as we will see) binary extensions.

Note also that we don't include version numbers in the directory name of each library. This convention is quite deliberate, and the rationale is both aesthetic and practical:

- the package version number is already encoded in both the pkgIndex.tcl and package Tcl scripts
- most applications only need one version of a package
- it is easier to upgrade to a later version (just by replacing files) without the need to rename directories
- in the unusual situation where there are multiple versions of the same package in an application, the directories of the older version(s) can contain version numbers, making it obvious which is the current version

Now we need a small modification to the hello.tcl script to use the new facilities. We'll create a few buttons this time

```
package provide app-hello 1.1
package require Tk
package require gbutton
set buttons [gButton #auto .]
$buttons new "Hello" bell
$buttons new "Bonjour" bell
$buttons new "Hola" bell
```

And now we wrap and use this as usual

```
$ sdx wrap hello.kit
```

One other point to note is that the Starkit is compressed - for example on Unix we see the following sizes

```
$ du -bs hello.vfs hello.kit
64000   hello.vfs
8192    hello.kit
```

So, adding packages to a Starkit is simply a matter of placing them under the **lib** directory, and invoking them in the usual way.

Binary extensions (i.e. platform specific shared libraries contained in packages) can be used under Starkits in the same way they can be used in any Tcl script. There is only one constraint - the extension must use the Tcl Stubs facility [12] so that it can be dynamically loaded into the TclKit interpreter.

The main downside is that binary extensions make a Starkit platform specific. But with a little care, it is possible to construct a cross-platform Starkit containing binary extensions for multiple platforms.

4.4 Multi-platform binary extensions

For example, if we wanted to use the Tktable [13] extension in a Starkit that must be deployable on Windows and Linux, we would first create the **lib/Tktable** directory, then subdirectories for Windows and Linux and place the appropriate shared libraries into each:

```
Tktable
|-- pkgIndex.tcl
|-- tkTable.tcl
|-- Linux
|   `-- Tktable.so
`-- Windows
    `-- Tktable.dll
```

Then we need to replace **pkgIndex.tcl** with one that will load the appropriate one for the current platform:

```
set platform [lindex $tcl_platform(os) 0]
set lib Tktable[info sharedlibextension]
package ifneeded Tktable 2.7 \
     [list load [file join \
          $dir $platform $lib] Tktable]
```

If the extension isn't stubs-enabled and you have access to the source, you can use CriTcl [14] to generate a cross-platform package ready for inclusion in a Starkit.

TclKit is designed to be the platform specific part of the application, and the Starkit the cross-platform part. Including libraries for multiple platforms is one way of preserving this distinction (and much preferable to building a custom TclKit that contains additional libraries). But this can become impractical if there are several platforms, large extensions or a large number of extensions.

One alternative approach is to deploy using a platform specific Starkit (i.e. a Starkit with only the libraries for a particular platform). This reduces the size, but at the cost of losing the cross-platform ability.

Alternatively, you could separate the application into two Starkits - a platform specific part and a cross-platform part.

Using this latter approach is quite simple to implement - , TclKit allows a script to source a Starkit:

```
   set platform [lindex $tcl_platform(os) 0]
   source $platform.kit
```

Then, compiled extensions for Linux would be stored in a Starkit called Linux.kit, for Windows in Windows.kit, and so on. To do this we create a platform specific Starkit containing all the libraries we require for our application. In this case we'll use Tktable and the tDOM[15] XML engine.

For example, the Linux.kit directory structure would look like the following

```
   Linux.vfs
   |-- main.tcl
   `-- lib
       |-- Tktable
       |   |-- Linux
       |   |   `-- Tktable.so
       |   |-- pkgIndex.tcl
       |   `-- tkTable.tcl
       `-- tDOM
           |-- Linux
           |   `-- tdom.so
           |-- pkgIndex.tcl
           `-- tdom.tcl
```

The Windows.kit directory structure would obviously be similar. Note that we have retained the separate Linux subdirectory, since this allows us to more easily merge these separate Starkits back to one cross-platform one at a later stage.

The **main.tcl** is quite simple - since there is no application code it just needs to invoke "starkit::startup" to set up the auto_path correctly.

Finally, we need to modify the application Starkit **main.tcl** so that it sources the platform specific Starkit on start-up. In the following code, we assume that both Starkits are located in the same directory:

```
   package require starkit
   starkit::startup
   set platform [lindex $tcl_platform(os) 0]
   source [file join \
             [file dirname $starkit::topdir] \
             $platform.kit]
   package require app-hello
```

Assuming that the Starkits are in the same location removes the need for complicated schemes to locate each Starkit. For example, on Windows there is no need to create registry entries to record the location of the Starkits. Installation becomes a copy - and perhaps the creation of a link or shortcut from a user's desktop to invoke the application.

This same approach of storing multiple extensions in a single Starkit has been used to implement Kitten - an experimental collection of Tcl/Tk extensions.

4.5 Kitten - a collection of binary extensions

Kitten[16] started as an experiment to build a "Batteries Included"[1] collection of extensions for use with Starkits.

Kitten was named because it originally contained ten extensions[2]. Now it contains many more, but the name has stuck (and the alternative KitFiftyFive doesn't exactly roll off the tongue).

Despite being experimental (and having quite a few rough edges) Kitten has been surprisingly well received. Perhaps this is because of the promise it holds for developers:

- Kitten is useful during the development phase of an application, for it frees the developer from keeping a copy of each extension within the application
- it can update itself from a central repository

The latter is possible because of the Virtual File System layer, and distinguishes Starkits from other wrapping schemes. More on this later.

But there is another technique for deploying platform specific code - combine TclKit, the Starkit and any compiled extensions into a single executable - called a Starpack.

4.6 Starpacks

A **Starpack** is a special version of a Starkit that combines a Starkit with a TclKit runtime into a single file.

Starpacks are standalone executables which run out of the box, making them even easier to distribute and use than Starkits. This convenience does introduce a number of trade-offs:

- Starpacks only work on the platform for which they have been built
- Starpacks cannot modify themselves
- Starpacks must be updated as a whole, and are (much) larger than most Starkits

However for "consumer" platforms such as Windows and Macintosh, Starpacks are more convenient - the user only has to download a single file.

And they have one other advantage - since the TclKit is self-contained there is no risk that the application code will be incompatible with future versions of Tcl/Tk.

Note that since Starpacks use the same packaging mechanism as Starkits, their content can be listed and extracted with the SDX utility. The main difference is that the "header" (i.e. the piece of code that is executed on start-up) is a large binary executable file, and that the files stored inside include all the standard Tcl/Tk runtime support files.

[1] The term "Batteries Included" is used within the Tcl community for a comprehensive Tcl/Tk distribution that comes "out of the box" with many add-on components. See the Batteries Included Wiki page at [17] for more details.
[2] The alternative name of TenTcl was also considered

4.7 Constructing a Starpack

Constructing a Starpack is quite simple, and only marginally different from build a Starkit.

If we go back to our hello.tcl example, to create the Starkit we used the following commands

```
   $ sdx wrap hello.kit
```

If we want to create Starpack for Windows, we need to tell sdx which TclKit version to use for its run-time interpreter. In this case, we'll use **tclkit-win32.upx.exe** - which is the UPX compressed version of TclKit for Windows.

```
   $ sdx wrap hello.exe \
           -runtime tclkit-win32.upx.exe
```

There is one restriction though - you can't specify the same TclKit file as the one which is used to run sdx (since it is already opened by the operating system when it runs sdx). Just create a copy and refer to that.

We end up with a self-contained Windows executable **hello.exe**. The size of this file is less than a megabyte - which is quite reasonable when you consider it includes a complete Tcl/Tk runtime environment.

As you can see in the above example, Starpacks **for** any platform can be built **on** any platform. It is possible to build a Windows Starpack on Linux (as above), or a Mac Starpack on Windows, etc.

A convenient technique is to use a Starkit during development of an application - e.g. for distributing interim releases amongst development or testing staff, and then wrapping the application as a Starpack for deployment beyond the development organisation. This has the advantage of using smaller, cross-platform Starkits until the application is released, and then a "sealed" Starpack subsequently.

Starpacks have been used to deploy a number of open source and commercial applications, including:

- NewzPoint[18] - an information browser for Windows written by Michael Jacobson
- TclTutor[19] - a computer aided instruction program for learning Tcl written by Clif Flynt

But the best example of a Starpack is TclKit itself - which is simply a Starpack without application code.

5 TclKit

TclKit is the platform specific part of a TclKit/Starkit deployment.

TclKit combines a number of extensions and libraries in a single executable file:

- a complete Tcl/Tk distribution
- the IncrTcl object oriented extension for Tcl [20]
- the MetaKit database library and the Mk4tcl interface [21]
- the Zlib compression library [22]
- the TclVFS package [23]
- the starkit package - the Starkit runtime package
- the UPX executable compressor (on Windows and Linux) [24]

Being a complete Tcl/Tk distribution, TclKit can be used as an alternative to the usual **tclsh** or **wish** commands. To use as a Tcl shell, just run as you would tclsh. To use as a Tk shell, add "package require Tk" to load Tk and TclKit is equivalent to wish.

And being a single file, TclKit is perhaps the easiest way to get a Tcl/Tk environment onto any particular platform. Versions are available for over a dozen platforms - making it the most portable Tcl/Tk distribution available.

Until recently, TclKit was kept current with the most recent development versions of Tcl/Tk. This meant monthly builds that tracked the Tcl/Tk 8.4 alpha and beta releases. Subsequent to the release of Tcl/Tk 8.4, the TclKit focus has changed to stability - builds will be less frequent and will only track official Tcl/Tk releases or to fix significant bugs.

Although TclKit is cross-platform, a deliberate effort has been made to limit the included modules to generally useful facilities. Whilst it would be tempting to include other modules and make TclKit more of a "Batteries Included" distribution, this would increase its size (perhaps significantly) - this making it less practical for use in network based deployment. And, as we have seen, it is easy enough to add compiled extensions to Starkits or to wrap them into a Starpack.

Having read this, one might legitimately ask "why **IncrTcl** and why not one of the other object oriented extensions?" . The first part is easy enough to answer - an object oriented extension can make Tcl code much simpler and more maintainable. This is particularly true of event driven GUI code, where it avoids the need to carry around a lot of context, or pollution of the global namespace. Any object oriented extension for Tcl would have done but IncrTcl is well established, relatively stable and adds only around 50Kb to the size of TclKit.

Even with Tcl/Tk and its extensions, the size of TclKit is still quite small. This is, in part, because all the runtime scripts are compressed using Zlib. But on Windows and Linux the size is further reduced by compressing all binary code using UPX (TclKit for Windows is less than 1 megabyte in size). Typically TclKit plus a substantial application fit on a single floppy disk.

6 Advanced topics

6.1 TclKit/Starkit architecture

As we have seen, TclKit is nothing more Tcl/Tk, several extensions plus a small amount of startup code.

TclKit consists of three parts:

- a large binary prefix - i.e. basically a wish executable statically linked with the above extensions
- a Metakit dataset that contains all the scripted components of the above packages (and is equivalent to the contents of the TCL_LIBRARY directory) which is mounted using the Tcl VFS, allowing it to look like a normal filesystem directory.

A Starkit comprises

- a small Tcl prefix containing startup code
- a MetaKit dataset containing all the files inside the Starkit.

When you “run” a Starkit or Starpack the underlying operating system launches the code contained in TclKit’s large binary prefix.

This, in turn, starts interpreting the small Tcl prefix in the Starkit, which performs the following steps:

- the small Tcl prefix asks the large binary prefix to re-open the Starkit, but as a MetaKit database (it does this using the **starkit package** located in the MetaKit database within TclKit).
- the starkit package opens the Starkit to be run - using the TclVFS package (also stored in the TclKit MetaKit database) to "mount" that file,
- the starkit package then sources the **main.tcl** file in the Starkit’s VFS.

At this point main.tcl is in control and does whatever the application developer specified.

Although Tcl's Virtual File System accounts for most of the transparency of Starkits - there are a few VFS limitations that should be recognised:

- when loading a shared library, Tcl has to copy it out to a real file (and then clean up later)
- on Windows, shared library clean up must happen on exit, but after serious errors this might not be done properly
- commands launched from *exec* and *open pipe* are not able to look inside VFS mounts
- on Windows, *cursors* can not be used from VFS (make a temporary copy to a real file)
- startup can be slower, because scripts are stored in compressed form by default (but also note there are circumstances where startup may be faster, because many open/close calls are avoided)
- memory use is higher when the Starkits VFS is modified, because the underlying MetaKit database collects changes between commits (the commit timer fires every 5 seconds by default)

6.2 Database management using MetaKit

As mentioned, TclKit also includes **MetaKit** - an embedded high-performance database package. As well as being used to hold the Starkit VFS, MetaKit can be used by developers in their applications. It fills the gap between flat-file, relational, object-oriented, and tree-structured databases - supporting relational joins, serialisation, nested structures, and instant schema evolution.

MetaKit is accessed from Tcl using the Mk4tcl package. Details on using Mk4tcl are outside the scope of this paper, but Mark Roseman has produced an excellent tutorial on using Mk4tcl [25].

For the more adventurous, Matt Newman produced an object oriented interface between Tcl and MetaKit. This (as yet undocumented interface which has been nicknamed Mk4too) exposes more of the MetaKit functionality at the Tcl level. The best place to find out more is to look in the MetaKit source for the file mk/tcl/mk4too.cpp.

For more on MetaKit see the MetaKit home page at [21].

6.3 Using Zlib for compression

The **Zlib** package is used by TclKit to read compressed Starkits but it too is available to developers.

Given that the contents of a Starkit are already compressed, one common use for Zlib is to compress a client/server communications protocol by adding just a few lines of code.

For example, the server side of a client/server application might use something like the following to send data to the client:

```
fconfigure -buffering full \
     -translation binary $client
...
proc send_to_client {client txt} {
    set data [zlib compress $txt]
    set len [binary format s \
                [string length $data]]
     puts -nonewline $client "$len$data"
     flush $client
}
```

Similarly, the client side would receive it using something like the following:

```
fconfigure -buffering full \
     -translation binary $server
...
proc receive_from_server {server} {
    binary scan [read $server 2] s len
    if {$len} {
        set data [read $server $len]
        set txt [zlib decompress $data]
    } else {
        set txt ""
    }
    return $txt
}
```

6.4 Using the VFS from scripts

Earlier versions of TclKit used Matt Newman's Tcl-only **VFS** implementation (which overlayed parts of the Tcl I/O system). Now that C based VFS support has been added to the Tcl core in Tcl 8.4 (see TIP #17[3]) the change was made to use Vince Darley's TclVFS package to expose the VFS layer so that it can be used from Tcl scripts.

The TclVFS facilities are also available to application developers - and support a wide range of virtual filesystem types. For example, a script can mount .zip archives, ftp sites, http sites, webdav remote disks, MetaKit databases, and even mount Tcl's proc namespaces as a filesystem.

For example, a zip file can be mounted as follows:

```
% package require vfs
% vfs::zip::Mount foo.zip foo.zip
% cd foo.zip
% glob *
```

or an ftp site

```
% package require vfs
% vfs::urltype::Mount ftp
% glob -dir ftp://ftp.tcl.tk *
```

More details about the VFS and its use can be found at the Tclers Wiki pages for VFS [8] and TclVFS [23].

6.5 Self updating Starkits

As we have seen, one of the key benefits of Starkits is that their internal VFS can be modified at run-time - they can even be self-modifying and self-updating. There is no risk of an application modifying itself and accidently damaging the file, because MetaKit supports transactions. Even a power failure will not damage a Starkit, it will simply revert to a previously committed state (no repair is ever needed: this is fully automatic).

This can be used to make a Starkit self-updating- so it can update itself when a later version is available.

This can be as simple as replacing files within the VFS, or it could be a more sophisticated scheme such as one based on an incremental file transfer algorithm (e.g. rsync[27]).

Another interesting consequence of Starkits being updatable is deployment over various media. For some time now Starkits have been used in applications where some part (and sometimes all) of the application scripts were distributed over a network in a client-server configuration at run time.

Taken to extremes, an application can be deployed as a small "placeholder", which on first starting (after installation) simply downloads all the necessary scripts, saving them locally inside the Starkit for subsequent use. One commercial application uses this approach to provide a user interface that evolves over time - based on the user interaction - but with no involvement from the user apart from establishing a network connection.

[3] TIP #17 Redo Tcl's Filesystem - specifies modifications to the Tcl core to allow non-native filesystems to be plugged in [27]

6.6 Adding help to a Starkit

You can use the **WiKit** package [28] to add read-only documentation to a Starkit. WiKit is a Tcl implementation of a Wiki [29] - a collaborative authoring tool. When used as a help system the documentation file is contained within the Starkits VFS and compressed along with other scripts and data.

To add WiKit to a Starkit, download the Wikit StarKit[30] and use it to create the documentation file. By convention, documents are stored in a **doc** directory under the Starkit VFS:

```
$ mkdir mykit.vfs/doc
$ wikit mykit.vfs/doc/mydoc.doc
```

Note that Wikit contains its own help, and so is a good example of a Starkit help system.

Next, unwrap Wikit and copy the **autoscroll**, **gbutton** and **wikit** packages from the lib directory to your Starkit lib directory.

When required, your application can display the documentation using the following commands:

```
package require Wikit
Wikit::init [file join \
         $starkit::topdir dir mydoc.doc]
```

A common approach is to have your Starkit display the help documentation when it is started with no arguments. If you do this, the Starkit should be able to output console and graphical help, thus:

```
if {[llength $argv] == 0} {
  if {[catch {package require Wikit}]} {
      # ... output console mode help
  } else {
      Wikit::init $path_to_wiki_datafile
  }
}
```

You can even add a help Wiki to a package Starkit (such as Kitten). The **starkit::startup** package returns an indication of how the Starkit was launched:

- starkit - called from a Starkit
- starpack - called from a Starpack
- unwrapped - called from an unwrapped Tcl script
- sourced - the Starkit was sourced by another script

We use this in the main.tcl to detect if the package Starkit was launched by a user (or a program), or sourced by another Starkit:

```
package require starkit
if {[starkit::startup] eq "sourced"} return
package require Wikit
Wikit::init [file join \
        $starkit::topdir dir mydoc.doc]
```

Note also that there are two additional optional parameters to Wikit::init. The first specifies whether the wiki should be read-only, whilst the second specifies the window into which the Wikit window should be packed (this defaults to the Starkit Tk toplevel window).

6.7 Code privacy

Starkits are particularly suited to Open Source projects - combining both source and executable into a single file that is easy to download and evaluate. Since Starkits are compressed the casual browser doesn't see the code, although it is relatively simple to unwrap them.

But for commercial applications there are times when you want to hide code such as proprietary algorithms, or licensing schemes.

Although there isn’t yet a standard mechanism yet to do the encryption or obsfucation, there are a number of approaches that people are using successfully to provide code privacy

One approach is to encrypt some of the Tcl scripts, and implement a command to decrypt them at run time. This would involve a small C extension that performs decryption and key management. This approach has been used in at least one commercial application with good success. But also note that you also have to be careful about other issues like the “send” command being used to introspect the application and view source.

If you intend to implement such an extension it is worth looking at CriTcl[14] which allows you to embed C code in Tcl scripts and transparently compiles it for you.

But perhaps the most practical solution is to distribute Tcl bytecodes - the intermediate “virtual instruction set” used by the Tcl interpreter. Mark Roseman and a few other people have been experimenting with this using the bytecode writing/loading facility from TclPro[5]. The **procomp** command is used to generate bytecode files from Tcl source and this is stored in the Starkit VFS. At run time the **tbcload** command (or its underlying library) is used to load and run the bytecode files from within the VFS.

Eventually there is likely to be a general solution based on this approach.

6.8 Starkits and standard Tcl

It is possible to unwrap a Starkit even when TclKit or SDX is not available - using a pure-Tcl script called **ReadKit**[31].

ReadKit can unpack (or just list the contents) of any StarKit without relying on TclKit or MetaKit.

To actually uncompress files, it needs either the Trf[32] or the Zlib extension. If these are not available ReadKit can convert a Starkit to a ZIP archive, which can then be unpacked using widely available utilities (such as WinZip on Windows).

If Zlib was available in the Tcl/Tk core (or part of a standard Tcl distribution) then Starkits could be come the default vehicle by which Tcl/Tk code is distributed - avoiding platform specific formats such as tar and zip.

And the starkit package could be modified to use ReadKit concepts to allow it to work within a standard Tcl interpreter (albeit without MetaKit underneath nor the ability to update the VFS).

6.9 Installation options

Starkits and tclkit are designed to be installation-free - one only has to copy or downloads individual files and they immediately become usable.

One important consequence is that Starkits can make a minimal impact on the target machine: there are no registry settings (unless the application introduces them), there are no files strewn all over the disk, and there is no need to have super-user privileges to start using a Starkit.

Deployment using Starkits can be summarised as:
- installation involves a single **copy**
- uninstall is just a single **remove**

The main benefits of this approach are:
- applications can be used out-of-the-box
- applications can be launched from CDROMs and read-only servers (useful when evaluating an application)
- applications do not break because some file was (re)moved
- applications do not interfere with other packages
- applications can easily be moved to another computer
- there are no version dependencies or conflicts (for example, no “DLL hell”)
- backups are easy - just copy the Starkit
- removing applications is easy and quick

But in some situations, the more familiar approach of a traditional “installer” may be appropriate (even if it does nothing more than copy a Starkit/Starpack to a local filesystem).

There are a number of approaches to this:
- wrap the Starkit/Starpack with a simple installer built using a tool such as Vise, InstallShield or a Tcl based installer such as InstallBase[33]
- provide a custom Starpack to perform the install
- make the application Starkit smart enough to install itself

The self-contained nature and smaller size of the last approach is definitely advantageous (especially when using network based distributions). To implement this, the Starkit can look for a copy of itself in a known location and, if not found, start the installation process.

Alternatively, if using a Starkit you can use the ability of a Starkit to update itself and store a file inside the Starkit to indicate that it has been installed. This way the Starkit can check to see if it has been installed and, if not, start the install dialog.

6.10 Safekit and chroot jails

Sometimes you want to run Tcl scripts in a secure environment - that is, one where the underlying filesystem cannot be touched.

This can be achieved with careful use of the Tcl Safe Interpreter facility [34] - which allows untrusted Tcl scripts to be run safely, and provides mediated access by such scripts to potentially dangerous functionality.

But there are times when you want to allow access to external files, but restricted within a defined areas of the filesystem (for example, in a CGI application).

This can be achieved on Unix using the **chroot** (change root) system call to block out all access to the local filesystem. Since TclKit is a static executable and carries a complete runtime with it, there is no need to access the underlying filesystem.

To achieve this we use a small C wrapper called **Safekit**. [35]. This wrapper launches TclKit after calling **chroot()** to limit access to the current filesystem subtree. This approach guarantees that scripts run within this context cannot access a file outside this environment, not even through external programs.

Safekit is just a small C program that must be installed "**setuid root**" (since the chroot system call can only be used by the super-user). On startup, it does a chroot("."), and then reduces permissions to the current user's normal ones. Then, TclKit is launched, passing all arguments on to it.

```
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char** argv)
{
   if (chroot(".") != 0)
       return 1;
   if (seteuid(getuid()) != 0)
       return 2;
   argv[0] = "/tclkit";
   if (execv(argv[0], argv) != 0)
       return 3;
   return 0;
}
```

To use this sandbox you need to place all scripts and data that are needed in a single directory area, along with a copy of TclKit and the Safekit wrapper. Nothing else is needed, provided that TclKit and Safekit are both compiled as fully static executables. If extensions are to be dynamically loaded, you will also need to create a .../lib area with all necessary runtime shared libraries. For example, to run "myscript.tcl" safely, set up something similar to the following:

```
safeplace
|-- myscript.tcl
|-- safekit
|-- tclkit
`lib
```

Then, change the permissions on Safekit to be setuid root:

```
# chmod -rwx safekit
# chown root safekit
# chmod u+s go+x safekit
```

As an example, we'll use the following myscript.tcl script:

```
puts "pwd = [pwd]"
catch { exec ls } err
puts "exec ls -> [split $err \n]"
puts "glob * -> [glob *]"
cd ..
```

When run with a normal tclkit, we see the following:

```
$ ./tclkit myscript.tcl
pwd = /home/steve/safeplace
exec ls -> myscript.tcl safekit tclkit
glob * -> safekit tclkit myscript.tcl
pwd = /home/steve
```

However when run with the Safekit wrapper the script we see that we can't get outside of the current directory and commands like **ls** are not available:

```
$ ./safekit myscript.tcl
pwd = /
exec ls -> {couldn't execute "ls":
             no such file or directory}
glob * -> safekit tclkit myscript.tcl
pwd = /
```

7 Repositories

Wouldn't it be nice if one could go to a website, pick a couple of packages, select a couple of platforms, and end up with a Starkit (or a few Starpacks) to which only the application-specific parts need to be added?

Much of this is possible today with Starkits and TclKit, albeit you have to do it manually. Pick the extensions you need, download TclKits for the platforms you are interested in, add your application code, and wrap it all up into a set of deliverable executables.

The missing piece is convenience.

Developers shouldn't be assembly-line workers - they should be factory managers. In such a world, a developer would specify the goal, define the required components and fill in the missing pieces (i.e. the application specific code). It should not be a totally manual process, since much of this can be automated. This end goal has been dubbed "SEAL" .... the **S**tandalone **E**xecutable **A**ssembly **L**ine.

A first tentative step in this direction has been taken - an archive is being created to help simplify the task of distribution of Starkits and Starpacks. This has been called **The Starkit Developer Archive** (or SDarchive) for now [36]. The motivation is not just to collect binaries (and so end up with yet another repository of stale and unmaintained data) but to work towards a structure which can both technically and socially maintain itself for a long time to come.

The initial aims of SDarchive are modest - provide a useful collection of Starkits. But also to build on this and leverage the Starkit model to make the deployment of complete applications, small utilities and individual packages simpler than ever before.

There are several steps to turn the initially passive SDarchive repository into what will ultimately be an assembly line for applications:

- a basic CGI interface to browse and make selections for downloading
- a web-based interface to manage submissions, updates, and build results
- a Tk-based local interface (over HTTP) to improve the interaction
- tying into the Tclers Wiki as a way to comment, make suggestions and contact authors
- aWiki-based way to tie documentation and package dependency graphs together

Although some of the above is just "blue sky”, it is fair to say that once the foundation is in place much progress can be made in a relatively short time. After all, this is all done through Tcl/Tk - scripting is not just the goal, it is also the enabling technology by which all this can be realised.

The longer-term goal of SEAL is to become a self-maintaining resource (through the contribution of developers an users). Prior experience with the Tcl'ers Wiki and the Tclers Chat show that when the parties involved have the right environment and incentives then something special happens - collaboration, synergy and community awareness take over. The economics of this equation are simple: "win-win" - while the costs to achieve this are indeed very low.

The SEAL resource won't come about overnight, but at the same time one has to conclude that all the technologies exist right now to make this eminently feasible.

8 So who uses this *stuff?

Starkits and Starpacks are used to deploy a variety of applications - both commercial and open source - in fields as diverse as 24x7 telecommunications projects, business automation, publishing and systems management. There are many thousands deployed world-wide.

One of the most interesting products is CareerDemon[37] - an automated career guidance assessment/reporting service, developed by a team led by Steve Blinkhorn.

CareerDemon is deployed via an installer that contains TclKit and the CareerDemon Starkit. The installer is usually downloaded over the Internet, which is relatively painless due to TclKit’s small size. As the user interacts with CareerDemon the user interface is modified, based on the user responses to a series of questions. New user interface scripts are generated as required by a central server and downloaded by the CareerDemon Starkit, which stores them within its own VFS. After the initial install the host operating system isn’t touched.

CareerDemon is one of those applications that has to work “out of the box”, do no damage to the host system and leave no traces when it is removed.

Starkits also help when supporting such installations. Supporting several thousand customers across multiple timelines could be nightmare, but it turns out to be relatively straightforward.

If a problem occurs (e.g. due to disk corruption) rather than trying to remotely diagnose the issues, support staff have the users e-mail their CareerDemon Starkit as an attachment. They are then able to unpack the Starkit, look inside for the problem, fix it and send it back.

This is analogous to a mechanic putting a car up on a ramp, looking for a problem, repairing it and returning the vehicle. Because everything bar the TclKit "engine" is in one file, it is easier to do return-to-base warranty work rather than trying to fix a problem remotely.

9 Conclusion

TclKit and Starkits are all about deployment - they lower the barriers to building easy to download, install and upgrade cross-platform applications

And they are an ideal way to deliver both open source and commercial applications implemented in Tcl/Tk. Using Starkits there need no longer be a distinction between source, packaged, deployed or installed applications.

Starkits fit in naturally with Tcl/tk development practices - allowing developers to work with familiar tools and paradigms. And they make deployment “safer” because it becomes part of the development process - not an afterthought with its own hassles and testing requirements.

So the Starkit story is that deployment has been solved.

And because of this Tcl/Tk has the potential to realise the elusive goal of "write once, simple deployment everywhere", perhaps more so than any other language platform currently available.

Acknowledgements

Thanks are due to Jean-Claude Wippler (the original author of TclKit and Starkits), Mark Roseman and Steve Blinkhorn for their ongoing collaboration on TclKit and Starkits, and for their assistance preparing this paper.

Also to Larry Virden, for his patient reading of the draft.

And, of course, the many giants on whose shoulders we have stood.

References

[1] The RPM Package Manager - http://www.rpm.org

[2] Apt - the Debian GNU/Linux package management facility - http://www.debian.org/doc/manuals/apt-howto/

[3] Vise - http://www.vise.com

[4] InstallShield - http://www.installshield.com/

[5] TclPro - http://www.tcl.tk/software/tclpro/

[6] freeWrap - http://freewrap.sourceforge.net/

[7] Wippler, Jean-Claude *Scripted Documents* - Proceedings of the 7th Annual Tcl/Tk Conference - http://www.usenix.org/publications/library/proceedings/tcl2k/wippler.html

[8] VFS - Tcl Virtual FileSystem - http://mini.net/tcl/VFS

[9] The TclKit Home Page - http://equi4.com/tclkit

[10] SDX Download Page - http://www.equi4.com/starkit/39

[11] gButtons - http://mini.net/tcl/3403

[12] The Tcl Stubs mechanism - http://mini.net/tcl/stubs

[13] Tktable - http://tktable.sourceforge.net/

[14] CriTcl - http://mini.net/tcl/CriTcl

[15] The tDOM XML engine - http://mini.net/tcl/tdom

[16] Kitten - http://mini.net/tcl/kitten

[17] Batteries Included - http://mini.net/tcl/2352

[18] NewzPoint - http://mini.net/tcl/newzpoint

[19] TclTutor - http://www.msen.com/~clif/TclTutor.html

[20] Incr Tcl - http://incrtcl.sourceforge.net/itcl/

[21] The MetaKit database and Mk4tcl package - http://www.equi4.com/metakit

[22] The Zlib compression library - http://www.zlib.org

[23] TclVFS - http://mini.net/tcl/tclvfs

[24] The UPX Executable Compressor - http://upx.sourceforge.net/

[25] Roseman, Mark - *MetaKit: Quick and Easy Storage for your Tcl Application* - available under http://www.markroseman.com/tcl/

[26] TIP 17 - Redo Tcl Filesystem - http://www.tcl.tk/cgi-bin/tct/tip/17.html

[27] Rsync - http://www.rsync.org/

[28] Wikit - http://www.equi4.com/wikit/38

[29] Wiki - http://mini.net/tcl/wiki

[30] Wikit Starkit - http://mini.net/sdarchive/wikit.kit

[31] Readkit - http://mini.net/tcl/Readkit

[32] Trf - http://www.oche.de/~akupries/soft/trf/

[33] InstallBase - http://installbase.sourceforge.net/

[34] Tcl Safe Interpreters - http://www.tcl.tk/man/tcl8.4/TclCmd/safe.htm

[35] Safekit - http://mini.net/tcl/3384

[36] Starkit Developer Archive - htt://mini.net/sdarchive

[37] CareerDemon - http://www.careerdemon.com