Using Starkits for Application Deployment

Most enterprise level software applications are complex affairs. They normally consist of the program code itself, plus libraries, runtime data, images, user configuration information, icons, help text, and often plenty more. Such complexity raises a number of problems, starting with issues of distribution and installation, through support and upgrades, to issues of security following uninstallation. Where an application is being distributed to hundreds of desktops in a large corporate roll out, these problems become enormous, and often present major headaches to development and administration staff.

Package management, using RPM, DEB, TGZ and the like, goes some way to alleviating some of these issues, but it doesn't solve them all. Users are frequently confused by dependency issues, and ensuring all sensitive user configuration files are cleaned up during uninstallation can be tricky. Package management provides very few features useful to an engineer trying to figure out why an application isn't running properly on a user's machine. More importantly, given the number of Linux distributions, the versions of those distributions in use at any one time, and the varying package management tools they all use, trying to maintain a complete set of packages for all versions of Linux is a nightmare. If the application needs to run on UNIX, Windows or Mac OS X as well, the problems are compounded.

What is required is a simple packaging method whereby an entire application could be packaged, distributed, installed and executed all as one single file. This one file would need to contain the program, its libraries (scripts and binaries where appropriate), its runtime data files, user configuration files, and everything else needed to run the application on any Linux distribution, and, where possible, on UNIX, Windows or the Mac.

Although this sounds extremely difficult for most languages, Tcl/Tk developers have actually been deploying applications in this way for several years. The technology behind this technique goes by the generic name of "Starkit", short for STand Alone Runtime Kit. This article will explain the concepts, benefits and uses of the method.

The Starkit solution

Starkits are a neat and innovative idea, and not only do they solve the immediate problems of distribution and deployment, they open up a whole range of new possibilities, ones which dramatically assist developers in application creation and maintenance.

A starkit, in a nutshell, is a complete application which is packaged up into a single file. "Complete application" here means everything the application needs to run. This includes the Tcl/Tk scripts, any binary libraries, the required data files, help files and so on. All of this information is packed into the Starkit file and transparently compressed or encrypted to save disk space or provide a level of code privacy. It is this Starkit which is distributed to the end user.

A Starkit file is not directly executable. In order to run it still needs a complete Tcl/Tk environment, and this is provided by what is called a Tclkit. A Tclkit is provided to the end user in one of two ways: either separately, or bundled together with the Starkit in one big executable file called a Starpack. Table 1 outlines the terminology.

Tclkit	The complete Tcl/Tk scripting language, packed into a single binary package.
Starkit	A Tcl/Tk application inclusive of scripts, libraries and compiled C code, bundled into a single file. Executed using the Tclkit binary.
Starpack	A Tclkit and a Starkit, bundled together into a single executable file.

At first glance the Starpack appears to be the ultimate solution. It’s a genuine single file solution – language and application all bundled together as one executable file. The truth is, however, that while Starpacks are a very popular method of distributing applications, the benefits of using a separate Starkit and Tclkit frequently outweigh the single file convenience of the Starpack. To see why this is, we need to look a little closer at the anatomy of a Starkit.

Inside Starkits

Normally when a developer is writing an application they will have a number of files scattered across a number of subdirectories. The top level directory might contain the starting point, perhaps called main.tcl. From that point one might see a lib subdirectory containing library code, an images directory containing a splash screen or some icons, a data directory containing runtime data files and so on. Such a structure might look like Figure 1.

```
application
├─main.tcl
├─lib
│ ├─script
│ │   connection.tcl
│ │   datastore.tcl
│ └─binaries
│   ├─linux
│   │   search.so
│   │   sort.so
│   └─win32
│       search.dll
│       sort.dll
├─images
│   splash.png
│   icon.png
├─data
│ ├─xml
│ │   skeleton.xml
│ └─text
│     outline.txt
└─settings
    default.ini
```

Starkits are a single file packaging of a complete filesystem hierarchy. They normally contain the exact hierarchy the developer uses to build the application. This is achieved using Tcl’s Virtual File System (VFS) facility, which is explained in detail in the sidebar. Since Starkit based applications run in the same filesystem hierarchy the developer uses to create and test the application, no repackaging is required, and nor is any extra testing needed to ensure the application still runs in a different environment.

Running Starkits

In order to run a Starkit, a Tclkit is required. A Tclkit is a binary executable, containing a packaging of the complete Tcl/Tk language compiled for the end user's platform. Precompiled Tclkits are available for many different platforms, including Linux, Win32, Solaris, HPUX, *BSD and others. The complete source is available for those running more obscure systems.

The Tclkit will take a Starkit file, mount the VFS inside it, then run the main.tcl script found in that VFS. The fact that the application is running inside a virtual filesystem is transparent; everything happens exactly as if the Tcl/Tk application were running normally in the machine's native filesystem.

The Tcl Virtual File System makes Starkits writable as well as readable. This means that the application can save its state and the user's configuration files inside its own Starkit package. Nothing important needs to be stored in a hidden directory in the user's home directory, or in a registry entry. As well as the obvious simplification of backups, this has a few important implications.

Firstly, the user can copy their Starkit file to another machine and immediately have their application and data running there. Moving an entire project from work to home requires copying one file onto a laptop or floppy disk.

Secondly, when a user experiences a problem with their Starkit based application, an engineer can request they email the Starkit to the support desk. The engineer can them examine the exact system the user is running – code, data, configuration and all. When the problem is fixed, the engineer can email the patched Starkit straight back to the user.

Thirdly, secure uninstallation is simple: simply delete the Starkit after use. There will be no sensitive data files or cache entries hidden away ready for investigation by prying eyes.

Finally, all this self containment means significantly more robustness. The user is unable to delete or uninstall any files vital to the application's well being, and the application is immune to the "DLL hell" sometimes caused by component upgrade.

Using Starpacks

As described previously, a Starpack is the bundling of a Tclkit and a Starkit all in one executable file. When a Starpack is executed, the Tcl/Tk language inside the embedded Tclkit is unpacked and given control. Once it has initialised, it loads the embedded Starkit – that is, the actual application – and executes it.

For many applications Starpacks appear to be the cleanest solution to issues of distribution and deployment. They come as a single file which saves the user having to download and install the Tclkit and Starkit separately. They are completely opaque, which means the user need never even know they are running a script. They also remove the (admittedly limited) threat of a future incompatibility between a Starkit and a previously installed and obsolete Tclkit.

For some consumer based applications Starpacks are indeed the best way to go, but there are some distinct disadvantages too. Firstly, they are much bigger than Starkits. Starkits only contain the program and its required data. Adding the entire Tcl/Tk language to the package adds a megabyte or so. Secondly, since the package is a compiled binary executable, Starpacks only work on one platform. Thirdly, and perhaps most importantly, Starpacks cannot modify themselves. The contents are read only, so saving configuration information inside the package is not possible.

One common middle-ground approach is to create an installable package for the target system – an RPM or DEB file, say - which contains just two files: the Tclkit and an skeleton of the application Starkit. Both of these files can be put somewhere distribution agnostic, such as /usr/bin. On first execution, the application Starkit copies itself to the user's home directory, and it is that copy which is actually executed and updated with user specific data. This approach is a fair compromise for many situations

Uses of Starkits

Starkits are a flexible technology. Some of the more common uses are:

- Application deployment, as discussed in this article. They are perfect for when an application needs easy installation and easy uninstallation leaving the host system untouched. On UNIX platforms, Starkits are an alternative for shar archives.
- Cross platform deployments, for when an application needs to run from, or be regularly moved across, Windows, Macintosh or UNIX platforms. Where necessary, a Starkit can contain binary files suitable for three different platforms, and the platform independent script part of the application can decide at runtime which file – DLL or shared object – to load.
- Complex deployments, like games, which often need significant intelligence to deploy properly. Instead of having a choice of 30 different RPM, DEB and TGZ files to download from, the user can download a single file which contains all the material and intelligence required to install and uninstall correctly. It is not unusual for a Starkit to contain an application which is basically written in C, but which has a Tcl/Tk front end simply in order to facilitate Starkit deployment and installation.
- Application demonstration. A Starkit and a Tclkit on a CDROM provide a complete application on a single disk. It can be run immediately with no installation necessary.
- Wikis and online help systems. A Starkit containing the TclHttpd "server in a script" program makes an excellent base for dynamic information systems like Wikis. Taking a complete Wiki application, server and all, from one machine to another on a single floppy disk makes for a compelling demonstration!
- Database systems. The Tclkit package contains a complete, built in database server system named MetaKit, embedded into the core Tcl/Tk language. A Starkit can use this server to store and query either data vital to its own execution (like a star map in an astronomy application), or data which the user provides (like names and addresses in an address book application). Conventional, external database servers can, of course, also be accessed.

Conclusion

Starkits are one of the features of the Tcl/Tk language which make it so compelling. They enable cross platform application execution in a way that is robust, simple and very easy to work with. In doing so, they solve so many problems associated with application deployment and execution, and open up so many new avenues of functionality, one has to wonder why all languages don't have something similar.

Sidebar - The Tcl Virtual File System

The Tcl Virtual File System (VFS) was added to the language at version 8.4.0, released in 2002. The aim of the VFS is to provide a “filesystem in a file” facility at the script level. The filesystem can contain scripts, libraries, runtime data, images, XML, configuration information, or any other kind of files, all contained inside a normal subdirectory structure. A VFS is “mounted” and “unmounted” as required.

Advanced Linux users will immediately recognise this concept as being the same as the Linux loopback mounting facility, and that’s exactly what it is. However, since it’s implemented in the Tcl language core, and exposed at the script level, the facility is fully cross platform and doesn’t require root system privilege in order to use.

However, there’s more to the Tcl VFS implementation than simple files. A complete API of file related operations is exposed at the script level, which means a programmer can mount any supported “filesystem”. Filesystems supported out of the box include ZIP, TAR, FTP, HTTP and several others. A developer can also reimplement – at the script level – the “open”, “close”, “read” and “write” commands (plus several others), in order for their “filesystem” to open a database, or a connection to a piece of hardware, or just about anything else they can think of.

Note that the VFS implementation exposes the “write” script level command, which means virtual filesystems are writable, as well as readable. Configuration data, or any other form of user specific data can be saved into the VFS, and that data is immediately available the next time the VFS is mounted.

A very simple, file based implementation of the VFS concept is behind the packaging of Starkits. If one unpacks a Starkit, a complete file system will be found inside it, containing directories full of scripts, images, data and so on. The Tclkit simply mounts this VFS and runs the main script inside it. Since all the data required to run the script is in the VFS too, no access to files on the external native filesystem is necessary.