Scripted Documents

Jean-Claude Wippler
Equi4 Software
jcw@equi4.com

ABSTRACT

Software used to be written as source code, which was then compiled and linked into a single machine-specific application program. With scripting languages, editable scripts are now executable without intermediate steps, but the dependency on lots of script files complicates robust deployment. A range of “wrapping” schemes are in use today to package scripts and extensions into a single file. The “Scripted Document” approach presented here goes further by offering a database-centric solution for packaging, installation, configuration, upgrades, as well as all application-specific data. An implementation for Tcl is described — using MetaKit as embedded database — with a summary of the experiences gathered so far.

Introduction

Scripting languages introduce the notion of quickly “gluing” together existing application and libraries, as well as custom-made extensions, in flexible ways. This paper takes a generalized look at the issues involved in deploying, maintaining, and evolving Tcl/Tk-based solutions. It shows how a more flexible solution, using an embedded database, can solve problems not addressed by current wrapping technologies.

One of the effects of moving away from monolithic compiled applications and towards scripted packages, is that the convenient single-executable packaging of the final “link step” is lost, and that many script files and extensions may need to be included in the installation process of the application. This is compounded by the fact that the script language implementation itself also needs to be installed, which also consists of many files and directories. As multiple scripted applications are installed, each with its own — often conflicting — requirements, as more platforms need to be supported, and when some of the files reside on shared file servers, installation all too often ends up becoming unwieldy.

The concept of “Scripted Documents” presented here offers a way to implement quick and conflict-free packaged deployment, while at the same time greatly simplifying upgrades and evolution of scripted applications. After a review of current approaches to packaging, the motivation and design of this new approach will be presented. Following are details of a Tcl/Tk implementation, examples of actual use, experience gained so far, and ideas for further refining and extending this concept. It will be shown that Scripted Documents are simple to implement, and solve a wide range of packaging problems encountered during and after deployment of scripted applications.

Why Package?

Packaged distribution is useful in a number of situations. It makes it possible to deliver software as a black box, to end users who want to use it without going into the underlying technology.

This is the case when a software product is sold as a user-installable system, or when it is intended to be used as a tool that should simply work “out of the box”. In these situations, the software is expected to change very infrequently. The more convenient and robust such a packaging scheme is, the more likely it is to work without adding support overhead for the developer / vendor.

These end-user situations make it attractive to fully encapsulate the scripting language, in a way that it does not depend on, nor interfere with, anything else on the end-user's system. Robustness also means that the packaged application should not break when users install further applications later - even if some of these packages are not as careful to avoid interference with the rest of the system. This is a reason why including all shared libraries and script files inside the packaged application can be an advantage, despite the increased disk space requirements.

Note that in other situations, particularly when system integrators are available to maintain and customize a variety of software applications on a large system used by many users, packaged distribution may not be desirable, as it prevents the integrator from making changes they feel necessary in their environment.

However, for the majority of cases, particularly with users running on Windows, Macintosh or personal Unix workstations, packaged distribution is the preferred solution.

Current Packaging Approaches

In this section we look at a number of current technologies used for packaging applications. Each of these must consider the following mix of components that comprise scripted applications:

1. A platform-specific *script language engine*, the "main interpreter", e.g. the Tcl binaries
2. Required platform-specific *language implementation libraries* (DLLs), e.g. libc.so
3. A standardized set of portable *library scripts*, the "runtime support", e.g. lib/init.tcl
4. A set of popular *core extensions*, either scripted or compiled, e.g. lib/Tk8.2/
5. Domain-specific *vendor-supplied compiled extensions*, e.g. lib/Mk4tcl/
6. *In-house extensions* to simplify common domain-specific tasks
7. Custom *compiled extensions* for performance and/or interfacing
8. The custom scripts representing the *main application* itself

Note that all but the last few items consists of generic code, which tends to be used in many different applications. It is code that tends to be stable, reliable and changes infrequently.

An important dimension is platform-independence. Scripts are portable: many files in the above list are identical for all platforms. When disk space was sparse, this used to be an argument to separate the scripts from the rest to allow sharing them (note that a fine-grained structure can also *complicate* maintenance).

There are several approaches today which simplify the process of packaging. All of them focus on initial deployment, not upgrading. Some act like an installer and unpack a large set of files on the target system; others "wrap" up everything into one file.

InstallShield / Wise

These represent the classical binary installers for Windows (RPM could be considered similar for Linux/Unix systems). They deliver their payload by creating a large number of files. This approach works well for a one-shot installation, and when subsystems are not shared. Once information is copied to central areas like the "c:\windows" directory and the registry, this approach can break down. It is not uncommon for Windows users to resort to a full machine re-installation after several months of active use.

tar x / configure / make install

This is the Unix-based source code install model (made popular by *GNU Autoconf*) for deploying applications on a variety of (mostly Unix-based) systems. It is *potentially* the most flexible approach, since it allows making changes at any level, but it has a flaw: it is oriented towards source-code deployment, which in turn depends on a *compiler installation* before it can be used. For scripted applications, installation of the proper compilation environment can become a major stumbling block. For end-users, this approach doesn't solve anything — it pushes the issues to another level.

Hybrid installation

Scripting languages like Tcl have long ago adopted a hybrid form, with the core implementation of the language being deployed through one of the two above means. They have been very successful by paying attention to detail and resolving issues to deal with many platforms and configurations. Tcl can be configured with static or shared libraries, on over two dozen different Unix systems, Windows, Macintosh, and more. Tk attempts to share as many configuration options with Tcl as it can (yet it cannot avoid introducing new ones, like where to find the X libraries on Unix). New releases and upgrades are simple to install — if you have the appropriate C compiler. This approach can still lead to conflicts when multiple versions of Tcl/Tk need to coexist on a machine.

All in all, the Tcl/Tk core is easy to deploy. The trouble starts with deployment of items 5-8 in the list given earlier. Extensions need to be told where Tcl/Tk has been installed, and they in turn need to be installed in a certain way for Tcl to find them. Given that these extensions usually come from different sources, the inevitable differences creep in. Getting a full set of packages, such as BLT, Expect, Itcl, Itk, and TclX installed is cumbersome. Upgrading one of the components to a new release can range from annoying to nearly impossible, if the extension does not properly address platform differences. Throw in a couple of useful (but not as widely ported) extensions, add support for both Windows and Unix, and all of a sudden you need to be an expert administrator. Not to mention the time it takes to install all this.

The following solutions are currently being used to package Tcl/Tk applications.

Stick with pure Tcl

This is a popular approach: install the Tcl/Tk core, and stay away from anything "non-standard". Write scripts, with perhaps a simple installation script to help users get started. This works well: no compiler dependencies, well-understood platform dependencies, and can be packaged as a few text files with a simple instruction on how to get started. Excellent examples are *TkCon* [8], and *BWidgets* [9].

Tcl2c

A refinement is to embed all scripts as C strings, and to rebuild Tcl/Tk as a *standalone shell*, which is easy to do with the *Plus Patches* [1]. This leads to a single executable combining all scripts, script libraries, and Tcl/Tk itself. This approach has several properties:

- For each platform to be deployed on, the packager needs a compiler and needs to repeat the entire wrapping process, even if the application itself consists only of scripts. A new “Wrap” mechanism is being developed, a refinement using a ZIP-compatible archive appended to the executable, to remove the need for re-compilation.
- It can lead to large executables, since every scripted application includes the complete Tcl/Tk core and all runtime scripts.
- It makes it possible to include binary extensions, by linking them in statically - though this again increases file size.

ProWrap and FreeWrap

ProWrap [3] (which is part of the TclPro product) and *FreeWrap* [4] collect all scripts and append them to a specially prepared standalone build of Tcl/Tk, i.e. at the end of the executable. On startup, the executable figures out how to get at those scripts and then reads and evaluates them as needed. According to ProWrap's web page documentation, it can pack and compress scripts as well as embed other files (but not shared libraries).

MkAppTcl

Richard Hipp's *MkAppTcl* [5] (and its predecessor *ET*) take a somewhat different approach, in that the perspective is shifted to having a main C/C++ application, which facilitates easily switching to Tcl/Tk requests. In a way, MkAppTcl maintains the model of *embedding* Tcl/Tk in C/C++, whereas the other approaches *extend* a scripted application with compiled extensions. The result is similar, with MkTclApp including a lot of support to ease the process of generating a standalone executable containing scripts, compiled extensions, and custom compiled code.

Application Logic vs. Application Data

Each of these approaches maintains a strict separation between *application logic* (scripted and compiled) and *application data*, which includes data files (textual or binary), database storage, and configuration files. In a way this is very useful, as the application is usually what the *developer* provides, and the data is what the *user* provides.

Yet this separation of code and data is at the heart of a range of some deployment and maintenance problems. New software releases no longer work with some documents, configuration parameters get detached from the application they apply to, documents can't be opened, applications don't know where their data is, and different application modules cause incompatibilities. These issues can be complex to deal with, requiring elaborate “environment settings” (Unix) and “registry settings” (Windows).

Let's face it: *real-world software deployment and maintenance is often a mess.* Installation introduces many problems, and leads to systems that are brittle.

A Different Perspective

As just described, wrapping consists of combining application logic (scripts) and application runtime support (language implementation) into a single executable, while application data remains separate.

Scripted documents choose a *different* separation, combining application logic and data into a single file, while separating out application runtime support.

The single application-specific portion combines all application logic plus data into a single database / file. It contains everything needed by the application, including all data that the application must manage after installation. This reflects a task-oriented and “document-centric” view, which is strengthened by the fact that scripted documents are *executable*.

Scripted documents represent *just* the application side of things. The scripting language implementation and all its support files are implemented as a separate and generic “runtime”. The next section describes such a runtime for Tcl, called “TclKit”.

Other systems exist where application logic is separated from a generic runtime, each as a single file. A popular example of these is in Java, where the runtime (i.e. the Java Virtual Machine) can read any number of “JAR” (Java Archive) files. Scripted documents differ from such solutions because the application specific piece contains both the application logic and modifiable application data.

This choice of separation into two components has several implications:

- Scripted documents are *portable,* allowing the vendor to deploy to users on any platform, simply by transferring the single file.

- Scripted documents store information *reliably*, when based on a transacted database package.
- Scripted documents have executable content, and look very much like a normal application to those who work with them. A scripted document represents the stored state of an application, as well as the active task while it is running.
- Scripted documents depend on *only* one other file, the runtime. This creates one new (very clear) dependency not present in wrapped applications, but removes the data file dependencies of wrapped applications.
- Runtimes are platform-specific, but have no application-specific content. There is exactly one runtime for each platform.
- A runtime requires no installation — it can reside in any directory on the execution search path.
- Un-installation is straightforward: remove the scripted document and the runtime.
- There is no need to install a scripting language, because the runtime is a self-contained system.
- Simple applications lead to tiny scripted documents. Scripted documents benefit from generic code shared through the use of a common runtime.

Effectively, *the scripting language core and its entire support system define an infrastructure, which scripted documents rely on in a well-defined way*.

Additional Uses for Scripted Documents

In the normal case, a scripted document will contain the logic for a single application, and all its data. However, it is possible to develop more powerful scripted documents, for use in a wider variety of situations. There are various ways to extend scripted documents:

- They can provide several user interfaces for the same application, for example a Tk based GUI version and a text based version, chosen at runtime depending on whether the user is able to run a GUI. The different versions of the interface manipulate the same data, stored within the scripted document.
- They can be used to package a number of related applications, with a simple file link or copy-and-rename causing them to behave differently. An example described later deploys a client/server system as a single file, making it easy to maintain consistent software.
- They can contain boilerplate utility code, activated from the command-line by specifying additional arguments. A generic startup script has been implemented for this, which also allows viewing and adjusting configuration parameters from the command line.
- They can take advantage of the transacted script storage to reliably update them selves through the network. Failure, or “rollback”, will cause the system to revert to its prior state.
- They can be as general-purpose or as application-specific as you need them to be. Creating a scripted document that manages other scripted documents is one way to introduce configuration, customization, and/or upgrade management.
- Writing a scripted document which acts as design tool / IDE for itself or for other scripted documents would be a powerful way to simplify their development. Taken one step further, a large application could consist of one file which combines the final application (and therefore all its scripts), as well as an *embedded* development environment for it - a fascinating option for long-term maintenance. Scripted documents are the natural equivalent of “executables” in traditional compiled software.

Scripted documents retain the installation advantages of conventional wrapping technologies, but also solve the nightmares associated with configuration conflicts and upgrades, while greatly simplifying multi-platform deployment for the application vendor.

Upgrades and Evolution

When scripts are stored in a database, you can do things with them, which have traditionally been done only with application data. One of the most interesting is upgrading or evolving the application code itself.

Scripts can be altered while the application is running, and by the application itself. As a result, software upgrades cease to be a special issue; it's just a matter of storing new data over the old and committing the changes. This is just as safe as for changes to application data, when the underlying database uses a transaction/commit model to apply all changes.

Consistent and fail-safe updating is a critical feature for fundamental changes such as script upgrades. This mechanism is actually more robust that ordinary software development: if an installer, compilation, link step, or even a simple editor save over an existing script fails, there is a slight chance that the result will be corrupted or left in a inconsistent - and inoperable - state. This cannot happen with scripted documents, as implemented here. The worst that can happen is the introduction of a logic error: upgrading to a new release which does not work. From a technical perspective, there is nothing which can guard against such errors, but scripted documents do offer a way to minimize their impact: make a quick copy of the scripted document, which is a single file, to create a backup.

An interesting way to deploy and manage the application-logic of scripted documents is through upgrades over the net. The *TclDist* [12] example described later on is a generic (and portable) scripted document, which just asks for the URL of a web server to obtain the application, and which knows how to synchronize its scripts to that server. Such automatic upgrading has proven to be a popular feature for end users (e.g. as demonstrated by such technology in recent versions of Windows).

One of the reasons why scripted documents can be so flexible is because the underlying MetaKit [11] database used in this implementation supports *dynamic schema evolution* - adding and altering data structures of a scripted document is instant, and relies on the same transaction security as every other change. When an upgrade of the application *logic* is stored, the application can quickly extend or otherwise alter the application *data* stored in that scripted document, and thus support added functionality of any kind.

Updating the Runtime

The discussion so far has only covered changes to scripted documents. The assumption is that changes to the TclKit runtime are far more infrequent, since it contains only stable and generic code. If backward compatible, TclKit can be simply replaced as the need arises. Scripting languages in general have a track record of remaining extremely compatible over the years, even as very fundamental additions and improvements get added-in. Tcl is no exception.

If new releases of Tcl/Tk or MetaKit come out which are incompatible in a critical way, a new runtime will be created with a different name. This will lead to a new generation of scripted documents using that runtime (TclKit2?). As with all scripted documents today, there is no interference between such different versions. TclKit intentionally has no version number information in its name. It is intended to be a stable component in the world of scripted documents, and should go through extreme lengths to maintain backward compatibility. The primary way to achieve this goal, is to let TclKit err on the conservative side by lagging new Tcl/Tk and MetaKit releases (other than bug fixes).

Implementation Details

Implementing scripted documents in Tcl is relatively straightforward. Several issues need to be dealt with:

Database storage

Scripted documents store all scripts and data in a single file, are portable across platforms, and must be failsafe to protect a scripted document even after a system crash. The *MetaKit* database library meets these requirements. The fact that it is a good fit for scripted documents is not surprising, since it is the result of several years of development by the author — with many related design goals.

Executable content

Scripted documents are executable, by using platform-specific tricks to create the illusion that they are applications. In reality they are documents, which invoke the runtime as part of the startup process. On Unix, this is done in the same way as for shell scripts: making the first line "#!/bin/sh" and setting the execute-permission bit. On Windows, scripted documents must use a ".tkd" (TclKit Document) file extension or a small batch file. On Macintosh, the "file creator" must be set to a specific code.

Mixed script/data evaluation

The final step is to prefix all data stored as an embedded database with a special bootstrap header script to launch scripted documents correctly. For this to work, Tcl has been slightly modified to "source" a script(ed document) without getting confused if there is additional data tagged onto the end of the script. The first part of a file is scanned for a zero byte - if found, script reading stops there. If not, the file is assumed to be a normal script and is read in as usual.

Runtime package

The counterpart of a scripted document is its runtime, i.e. TclKit. Based on the *Plus Patch* [1] version of Tcl/Tk, a standalone executable has been created which consists of the Tcl and Tk core, all necessary supporting scripts, as well as the Tcl-aware version of MetaKit, called *Mk4tcl*. The *Trf* [6] extension is also included, to give access to *zlib* [7], which is very useful for packaging.

Let's examine in detail what happens when a scripted document is launched (using an imaginary "example.tkd" document on Windows as case study):

1. User double-clicks the "example.tkd" scripted document
2. Windows users should associate ".tkd" files to the tclkit.exe file, so that the runtime starts up upon double-click
3. TclKit opens the example.tkd file, reads everything up to the zero byte, and starts evaluating that data as a Tcl script.

4. Here is a basic version (without script compression or error handling) of the bootstrap header script:

```
#!/bin/sh
# \
exec tclkit "$0" ${1+"$@"}
package require Mk4tcl
mk::file open doc $argv0 -nocommit
eval [mk::get doc.scripts!0 text]
return
```

   The first three lines are the standard Tcl'ish way of launching a script. These lines are only used under Unix, and skipped otherwise. The remaining lines initialize the MetaKit database in TclKit, reopen the scripted document as a database, fetch the "text" field of the first record in "scripts" as a string, and evaluate that string. If the script returns then break off (this is plus-patch specific; without it Tk enters an idle loop).

5. That's it, as far as the basic bootstrap into scripted document in Tcl is concerned! The first script stored in the database determines the rest.

Some comments:

- *Scripted documents rely on a very simple mechanism.* It's not much more than a single file / database storing one or more scripts, as well as any other type of data which needs to be stored. What makes them special is the way everything is packaged.
- The bootstrap process described so far is all there is to it. But the fact that all data now resides in a database, including the scripts needed to work with that data, and that this file is *modifiable by these same scripts* is what causes this to be an open-ended approach. You can make things as sophisticated as you like, by creating and including the appropriate scripts.
- All scripts in a scripted document can build on all of Tcl, all of Tk, and all of MetaKit - because all of these are always present. This means that as with any plain Tcl/Tk installation you have the power of scripting, networking, a cross-platform GUI, as well as a robust database, at your disposal. *That's a lot of infrastructure!*
- All changes to a scripted document are transacted, because this is how MetaKit works. Each scripted document can be used for large-scale and efficient demand-loaded data storage.
- On Unix, scripted documents can be used as command-line Tclsh-like applications, or as visual Wish-like applications. This is possible because the plus-patch version used in TclKit includes Tk as a run-time facility. A CGI application for web-server use need not activate Tk (and will not require X Windows) for example, while the same application can be also be used to run in fully GUI-oriented mode in other situations. Note that Windows cannot mix console and GUI modes, and that the Mac has no system console mode.

Startup Script

The first script in the database is what gets executed as the last step of the above bootstrap. Although one could store the main application script, normally a *command line interface* script (CLI) is stored there. This script examines command line arguments to offer a basic level of support for scripted documents. Among the functions it performs are:

- Adding or replacing database scripts using files or directories specified as argument
- Basic listing- and extraction facilities for scripts and other text-based information
- The ability to start up an alternate script by giving its name on the command line
- Defining a default startup script if no arguments are specified, or ignoring them
- Viewing and altering configuration options, which are also stored in the database
- A few utility procedures for scripts to easily access those configuration options

With no command line arguments, and if not specifically configured otherwise, the CLI simply looks for a script with the base name of the scripted document and executes it (if the scripted document is "example.tkd", the CLI would expect to find a database script entry called "example.tcl").

The CLI acts as an important safety net, in case a scripted document fails to start up properly (perhaps because a vital script was altered or deleted). As long as the first script is the CLI, one can examine the contents, restore scripts through the command line, launch scripts which validate or repair other parts of the database, and so on. In a way the CLI is like the *BIOS* of PC's, or the *bootstrap monitor* of embedded systems: rarely changed, but crucial during startup and recovery.

Note that scripted documents can only be "damaged" by scripts making improper changes and then actually *committing* those changes - system faults, crashes, even premature exits, will automatically revert the contents of a scripted document to the last committed state. Damage to scripted documents due to bypassing the database code is currently not detected, there are plans to implement checksums for some key data structures in MetaKit.

Using Extensions

With scripted documents, everything is fine... until you need to use dynamically loaded *compiled* extensions. These may be unavoidable to interface to existing code or to achieve acceptable performance for CPU-intensive tasks. Now, the ugly issue of platform dependence comes back — conflicting with the simple distinction used so far: a portable but *application-specific* scripted document and a platform-specific but *generic* runtime. The question is: where do you put these platform-specific extensions, in the context of scripted documents? There are a number of options:

Store the shared library on disk and remember path

This is the standard mechanism provided by Tcl. It can be streamlined by setting up *pkgIndex* and *auto_path* to automatically find and load extensions. The main problems of this approach are that different extension builds needs to be used on each platform, and that the cost of deployment and maintenance can be high.

Include the shared library in the scripted document

From a deployment standpoint, this is the preferred way. To be able to load such extensions, the package mechanism must be told how to extract the shared library to the disk, and then launch it. This solves the deployment and maintenance issue, but runs only on the platform corresponding to the library included in the scripted document.

Include several platform builds of the shared library

A refinement is to include several shared library builds, and to select the proper one for extraction at runtime. This extends the portability to all builds that have been prepared in this way, at the cost of increasing the size of the scripted document. Yet another refinement is to also include a Tcl-only version of the extension, which is used if no other build is suitable. The Tcl-only version could be a slower or more limited implementation, or a script that presents a clear explanation of why this functionality is not available.

Store shared libraries on an HTTP or FTP server

Given that the potential number of builds for compiled extensions can be very large, it is tempting to create a central place, where new builds get added over time. The package load mechanism can be similar to the above one, but instead of checking the scripted document, it would attempt to fetch the appropriate shared library from a remote server. This requires a trusted network environment.

Create "vendor-specific" scripted documents

For popular extensions, scripted documents could also be used to create a special "deployment package" for an extension such as Itcl/Itk/Iwidgets. It would have as only task to deploy those extensions, and to easily manage and upgrade them. Tcl scripts and other data are stored in this scripted document. In a way, this is an installer for that vendor's extension, which could take advantage of scripted documents to provide out-of-the box execution on any platform, demo's, documentation, an installation verifier, test suites, a cleanup facility, and perhaps net-based upgrades.

One last remark: TclKit supports "stubs", this is essential on some platforms to be able to load compiled extensions dynamically.

Practical Experience

The best way to summarize the experience with scripted documents so far, is: *they make deployment fun!*

CGI Applications

CGI applications are a great use for scripted documents. A typical Tcl-based CGI application consists of a variety of CGI scripts which must be installed. Of course, a Tcl interpreter needs to be installed. If it needs to store data in a database, a separate database extension must be compiled and installed. Finally, if the application contains pre-loaded data, all of those data files must be installed. That's a lot of separate pieces a webmaster needs to worry about to run your CGI application, particularly if they are not already using Tcl for their web site.

A simple example illustrates the scripted document solution. A CGI application implementing a simple bug-tracking system (written by Mark Roseman of TeamWave) started out having all the separate pieces identified above. It was turned into a scripted document with just a few simple changes:

- All scripts were added to a fresh scripted document with just the CLI code in it.
- The database access was altered to use the scripted document itself for storage, rather than an external MetaKit data file.
- A few configuration parameters were defined for easy customization.
- The system was extended to list some help text when not called as CGI process.

The whole process took perhaps half an hour. The result is a bug tracking system scripted in Tcl, consisting of just one file which runs on any system for which there is a corresponding TclKit executable (Windows, Macintosh, and several Unix'es). See the reference link to "BugCGI" [12] at the end to access / customize / use this system yourself.

WiKit

WiKit [12] is an implementation of a so-called “wiki-wiki web” — a tool which lets people enter and edit hyper-linked textual information over the net using nothing more than a web-browser. WiKit adds things like a search engine and a Tk user interface when used locally, and uses MetaKit to store all pages and the change history. WiKit includes about 2,500 lines of custom Tcl scripts. Several WiKit based systems have been in constant use since early 1999 (such as *the Tcl'ers Wiki* [14]); their biggest “drawback” being that it has become too easy to set up lots of them — not a good idea, in term of content management. Note how software deployment has become so simple and effortless, that it no longer matters...

TclHttpd

Matt Newman has created a scripted document containing Brent Welch's powerful tclhttpd server, thus creating what must be the most easy to deploy scripted HTTP server ever. Some additional scripting was introduced to make the scripted document behave as a normal file system for the server, and to allow merging contents from an external path with the files stored internally. There are a wide range of potential uses for this system, ranging from deploying a portable standalone documentation server to full HTTP-based client/server applications.

TclDist

As a first experiment in automatic script maintenance, a small generic bootstrap utility called *TclDist* [12] has been created as scripted document, which fetches a list of applications from the web. It then uses the *httpsync* [10] protocol to fetch the selected set of files that get inserted into the scripted document itself. The application that is created in this way is not only ready for use — it also knows how to check for new versions and how to update itself. This net-based mechanism turns out to be the easiest way by far to deploy scripted documents. It requires a trusted HTTP server and can perform efficient differential updates.

Large Scale Deployment

The first commercial project using scripted documents has been another excellent source of experience. For confidentiality reasons, this system will only be described here in general terms. It is a distributed application with two long-running processes used for unattended testing of specialized equipment, combined with two different user interfaces: a management interface used remotely by administrators to monitor correct operation and present reports and statistics, and a test-set interface to let a group of field engineers schedule ad-hoc and periodic tests to verify/stress certain parts of the system.

The whole application was built in pure Tcl/Tk plus MetaKit, and consists of two non-stop server processes (called STORE and CONTROL) and the two types of client applications, one of which is installed on about a hundred workstations (mostly Solaris and Windows NT). Due to the large number of machines on which the client packages needed to be present, and the fact that this project was at the start of a much more ambitious system with more modules and client application types to be added later, the project was an ideal context for deploying software as scripted documents.

The end result: a *single* scripted document of under 250 Kb, plus one TclKit runtime for each platform is all that is needed to deploy this elaborate client/server system. It consists of some 550 KB of Tcl scripts (roughly 14,000 lines of code, stored in compressed form), and it runs out of the box. The field engineers do not see Tcl, they simply see a single application, which they can store and use wherever they like.

The scripted document contains all application logic for all server and client processes, and adjusts its behavior based on the name under which it is stored. Copying the distribution file to a file called “STORE” sets up the database server. Copying it under the name “CONTROL” sets up the non-stop scheduler and equipment interface of this system. The same applies to both types of client applications in this project.

One further refinement makes this scheme complete: before the distribution is copied in this way, it is configured (through the standard scripted document command-line interface) with parameters which specify the central server host name and the port to use. As a result: all clients are pre-configured and know how to communicate with the main server. Since they are consistent, there is far less chance of mixing up incompatible versions of the different applications. Note also that all this “deployment” is portable, and that the only platform dependency is a check that the appropriate build of TclKit is present on the platform.

Upgrades and long-term evolution of this system is done by upgrading the STORE server, and then having all other processes synchronize their scripts to the server. This is automatically done whenever a client starts up, over the network, but it has so far only been carried out as an experiment. The goal is to make upgrades user-initiated but fully automatic, distributed by the STORE server, and to do this without bringing the system down, except when a few core components need to be changed. Further details still need to be worked out, but it looks like the current design will support this mechanism.

If you have ever deployed a system of such complexity, you will understand just how much administrative effort has been *avoided*, and why scripted documents are effective. Several features stand out in the above example:

Installation is a non-issue. Unless you call copying and renaming of one or two files "installation", it is clear that this aspect of deployment no longer matters.

Self-consistency. The combination of all application logic into a single file greatly reduces the chance of mixing up revisions and releases.

Net-borne upgrades. Whether on Intranet or Internet, the fact that scripted documents can safely update themselves from a central server makes upgrading easy, and can be automated as far as needed.

Data storage is implicit. The STORE server stores all application logic and data. Users of the system do not see this database as a separate entity in the system, it is simply “part of the server”. The embedded database can be fully inspected and altered through the general-purpose Mk4tcl scripting interface.

Self-diagnosis. Since everything now happens inside scripted documents, a range of deployment-related tasks can be handled in a more generalized way, by including some diagnostic scripts as part of the scripted document. An example of this is a simple “dump.tcl” script, which has been written to inspect the contents of a scripted document — any scripted document.

Long term backups. A backup of a scripted document is more useful than a traditional database or file/tree backup, because it is more complete: it includes the essential data, but also the application that manages that data. Since they are also MetaKit data files, and since MetaKit has maintained backward file format compatibility since its first release in 1996 and will continue to do so, the data remains accessible. If needed, a utility script could be added to export all data in XML format. This export potential is permanently tied to the scripted document, even when transported to another system or stowed away for a long time.

Scripted documents fulfill the promise mentioned in the introduction: they solve “a wide range of problems encountered during and after deployment”.

Future Work

The concept of scripted documents is simple. It took less than a week of work to make all the essential pieces play together. But what has been described so far only scratches the surface of how they could be put to use. Here are some ways in which scripted documents could be extended:

Generic self-development and self-management tools

This is the most obvious area of further development: tools to create, inspect, and alter scripted documents. As well as far more ambitious ones: an embedded IDE which lets you develop a scripted document with itself, a built-in visual design editor, generalized update-over-the-net tools, hooks for revision control, customer support embedded in an application, documentation viewers, an embedded debugger, and so on.

Getting rid of the compiler

The Achilles' heel of scripted documents is the shared libraries that need to be deployed as part of an application. A project by the author, called “Minotaur” [13], explores the usefulness of combining a high-performance portable Forth engine with scripting languages. One of the benefits is that tight loops and low-level functionality can then be written in Forth instead of C, maintaining the portability which makes scripting so attractive. Forth can also be used as glue to shared libraries at run time, removing one of the main reasons to create C interfaces.

It should be noted that scripted documents are not Tcl-specific, even though the current implementation is. There is no reason why the same concept could not be applied to Python, even shell scripts.

Conclusion

It takes more than just developing software to make software-based solutions work. This paper has presented a new concept for packaging and deploying scripting-based applications which changes the landscape of software installation, configuration management, but especially software upgrading and evolution.

The idea is to use a modular and component-based approach during development — an approach that matches the “gluing” nature of scripting well — and then to combine the pieces in easily deployed and out-of-the-box runnable “scripted documents” when delivering complete solutions. The redundancy of including some modules repeatedly in each software package is easily offset by the total clarity this mechanism introduces in the later stages of system evolution: deployment, maintenance, and upgrading.

The key difference between scripted documents and wrapping approaches are that scripted documents use a database to merge application logic and application data into a single file, while separating out the runtime. A sharp distinction is thereby made between the platform-specific *infrastructure* (the “runtime”) and the platform-independent *application logic plus data* (the “scripted document”).

Scripted documents solve the fragility problems inherent in separating application code and data. They greatly simplify multi-platform deployment for the application vendor, and introduce new opportunities for software upgrades.

This paper has shown how a simple mechanism can be implemented as a "TclKit" runtime for Tcl based on Tcl/Tk and the MetaKit embedded database, and how effective it ends up being in actual use. Refinements and future plans have been covered to highlight the current status and the potential of this approach.

Scripting languages are a major step forwards in programmer productivity because of their rapid application development style. Yet they fall short when it comes to the task of getting a software solution out into the hands of its users and walking away from it in a "problem solved, case closed" fashion. Scripted documents add that final step needed to make scripting not just effective to build with, but also very effective to deploy lasting turnkey solutions.

Acknowledgements

This paper originated from a suggestion by Matt Newman. It would not have been readable without the patience, feedback, and detailed input from especially Mark Roseman, as well as Matt Newman, Larry Virden, and Christian Tismer. I am grateful to each of them for their valuable advice and suggestions, and to Jan Nijtmans for extending his Plus patches to make Tcl work with scripted documents.

References

[1] The Plus patches, Tcl2c, and the new Wrap extension, *by Jan Nijtmans* http://purl.oclc.org/net/nijtmans/plus.html

[2] Tcl Extension Architecture (TEA) *by Scriptics* http://www.scriptics.com/products/tcltk/tea/

[3] ProWrap commercial wrapper, *by Scriptics* http://www.scriptics.com/products/tclpro/wrapper.html

[4] FreeWrap, a Tcl/Tk standalone, *by Dennis LaBelle* http://www.albany.net/~dlabelle/freewrap/freewrap.html

[5] MkTclApp embedding wrapper, *by Richard Hipp* http://www.hwaci.com/

[6] Trf transformation framework, *by Andreas Kupries* http://www.oche.de/~akupries/soft/trf/

[7] Zlib library, *by Jean-Loup Gailly and Mark Adler* http://www.cdrom.com/pub/infozip/zlib/

[8] TkCon console interface for Tk, *by Jeffrey Hobbs* http://www.purl.org/net/hobbs/tcl/script/tkcon/

[9] BWidgets user-interface widgets for Tk, *by Unifix* http://www.unifix-online.com/BWidget/

[10] Httpsync protocol, *by Forrest J. Cavalier* http://www.mibsoftware.com/httpsync/

[11] MetaKit database, *by Jean-Claude Wippler* http://www.equi4.com/metakit/

[12] TclKit runtime and the sample BugCGI, TclDist, WiKit scripted documents *by Jean-Claude Wippler* http://www.equi4.com/tclkit/

[13] Minotaur project, connecting Tcl, Python, and Perl, *by Jean-Claude Wippler* http://mini.net/pub/ts2/minotaur.html

[14] Tcl'ers Wiki, a collaborative web site for the Tcl community, *maintained by Jean-Claude Wippler* http://purl.org/thecliff/tcl/wiki/