CANTCL: A Package Repository for Tcl

Steve Cassidy
Centre for Language Technology,
Macquarie University, Sydney
E-mail: Steve.Cassidy@mq.edu.au

Abstract

For a long time, Tcl users and developers have requested some kind of coordinated package repository; CANTCL defines a standard web based interface to such a repository and provides a reference implementation. This paper will describe some details of the reference implementation and develop some ideas for applications that might be built on top of the capabilities of CANTCL.

1 Introduction

Many Tcl programmers would like access to a central, organised archive of Tcl extension packages both to store their own work and to get access to the work of others. Various attempts at setting up a traditional ftp based repository have not succeeded in gaining a central place in the Tcl world. This paper describes CANTCL, a proposal and trial implementation of an augmented repository which adds significant value to the simple file stores that have gone before.

2 TIP55

As a precursor to building CANTCL, some work needed to be done on defining what a Tcl package was and how it should be built and structured. The Tcl Extension Architecture (TEA [6]) describes a standard way of building C coded extensions which has enabled the inclusion of so many packages in the recent ActiveState Tcl distributions. However, TEA doesn't provide enough information on how to bundle up extensions for distribution.

TIP55 [1] attempts to solve two problems relating to package distribution: the layout of files inside a package bundle and the metadata format associated with these bundles.

2.1 Directory Structure

In order to be able to automatically install packages, some knowledge of how they are laid out is needed. Fortunately, Tcl has settled on a reasonably standard package structure where each package has its own subdirectory somewhere on the `auto_path`. Inside this directory is a file `pkgIndex.tcl` which contains Tcl code to make the package available by sourcing Tcl code or loading shared libraries.

The simplest Tcl package format then would be an archive file containing a snapshot of the package directory from a working installation. To install the package on another machine, only unpacking of the archive would be needed. This simple model is complicated when shared libraries are included in the package since we must then distinguish the platform or that these have been built for. The easy way out is to have different packages for different architectures but this precludes building multi-platform package archives. Fortunately it is relatively easy to put architecture specific files in a subdirectory and have the `pkgIndex.tcl` load the appropriate one at runtime. In order to make package directories a little neater, it's useful to put Tcl code in it's own subdirectory also; this can be thought of as code for the special architecture `tcl` and so Tcl only packages can be categorised in this way.

TIP55 adds two directories for documentation and code examples to this basic set. Documentation should be considered mandatory for any software being distributed to third parties and it would be useful to be able to define the format and naming conventions of package documentation. At the time of writing of TIP55 no single standard had emerged for writing Tcl documentation. Recently the `doctools` format has matured and is now widely used; it might be useful to suggest that a file `pkgname.man` be included for each package in the archive. This would allow package help to be linked in with the online help system on the platform, be that Unix man pages or html based help.

The suggested directory layout definied by TIP55 is then:

```
packagename$version
    + DESCRIPTION.txt  -- Metadata
    + doc/             -- documentation
    + examples/        -- example scripts
    + $architecture/   -- shared libraries
    + pkgIndex.tcl     -- package index file
```

The `pkgIndex.tcl` file may be omitted if this directory structure is adhered to as it can be automatically generated – in particular, if multiple architecture shared libraries are included, a `pkgIndex.tcl` file can be generated to load the appropriate version for the platform.

2.2 Metadata

Metadata is central to a usable package repository; it allows the repository to find packages based on various criteria and allows human browsers to decide whether to download a package or not. TIP55 defines a simple metadata format which is borrowed from other repositories such as CRAN [3] and Debian [4]. Also defined are a number of required fields and their meanings, for example *Creator* for the entity creating the package, *Require* for a package required by this package. Some of these names are taken from global metadata standards [5] and others are taken from other repositories or are defined just for CANTCL. This gives us some potential for interoperability with other repositories and indexing services. Metadata is stored in the `DESCRIPTION.txt` file in the top level of the package directory.

3 URI Interface

An earlier paper [2] describes some of the design considerations that have gone into the development of the web services interface to CANTCL. The result is a API based on accessing http URIs for the different services provided by the repository. All CANTCL URIs are relative to a base uri `http://purl.org/tcl/cantcl` which is a permanent URI which redirects to the CANTCL cgi script. The first path element following this names the action being performed on the repository (one of `package`, `browse` or `upload`). For brevity, the prefix will be elided from the examples below. The three interfaces provided by CANTCL are:

Download is implemented via HTTP GET to a `package` URI which contains information about the package name, version and format. For example:

`.../cantcl/package/installer0.4.zip`
`.../cantcl/package/PMG.kit`

Download of files from within a package is also supported:

`.../cantcl/package/PMG/tcl/pmgvar.tcl`

Browse/Search is implemented via HTTP GET to a `browse` URI. The default return type is HTML but this can be modified by the next path element (eg. `browse/xml` or `browse/text`). Query terms can be appended to the URI to match fields in the package metadata. For example:

`.../cantcl/browse?requires=tip55`
`.../cantcl/browse/xml?requires=tip55`

Upload is implemented via HTTP POST to `.../cantcl/upload`. The file uploaded can be any format supported by tclVFS, eg zip, starkit, tar file.

Any server implementing the above URI interface can be considered a CANTCL server. This allows for the possibility that on the server, packages are stored as files, inside a metakit database or in CVS.

4 Installing Packages

To install a package from CANTCL it needs to be downloaded and unpacked in some directory where it will be found by the default package loading mechanism. For Tcl only packages with an existing `pkgIndex.tcl` file, this is all that is needed. When the index file isn't present, it must be generated by running `pkg_mkIndex` with appropriate arguments to pick up platform specific shared libraries.

There may be systems where more work is needed for a package to be installed from an archive; for example it may be necessary to move platform specific libraries into a system directory. While a general purpose installer is unlikely to be able deal with this problem, a platform specific script should be able to take advantage of the well defined structure defined by TIP55 to perform the appropriate installation actions.

As mentioned earlier, another action that could be performed at installation time is the integration of package documentation with the standard platform help system.

This simple installation procedure is made possible by the well defined structure provided by TIP55; if we don't know where to find various package components such as shared libraries or help files, automatic installation becomes very difficult.

5 A CANTCL Implementation

The initial CANTCL implementation is made up of a number of packages which provide services relating to packages, the CANTCL cgi interface and client side package installation. The packages are described here.

5.1 Package `tip55`

The `tip55` package provides procedures for manipulating TIP55 compliant packages. The two primary areas of functionality are to provide an interface for reading and writing package metadata and for validating package structure. For example, `tip55::package_description` returns a pairlist containing the fields and values from the description file in a package directory and `tip55::validate_package` checks conformance with the TIP55 directory layout specification.

Since packages might be stored in the file system or inside some kind of archive, the `tip55` package provides a utility to work with any mountable archive as if it were a regular directory. The procedure `tip55::with_mounted_dir` ensures that a directory or archive is mounted while running a piece of code, for example:

```
tip55::with_mounted_dir $dir {
    set foo [glob -directory $dir *.txt]
    ...
} except {
    puts {Can't mount directory}
}
```

Packages can be stored in different locations inside an archive: eg. in a subdirectory inside a zip file or in a subdirectory of `lib/` in a starkit. To provide a clean interface to code manipulating packages, the procedure `tip55::foreach_package` is provided which executes code for successive packages inside an archive:

```
tip55::foreach_package pkgdir $dir {
    set desc [tip55::description_name $pkgdir]
    ...
} except {
    puts {Can't mount directory}
}
```

5.2 Package `cantcl`

The `cantcl` package provides most of the infrastructure for the CANTCL server implementation. This can be split into two parts: decoding URI requests and providing an interface to the repository.

`cantcl::decode_url` parses the request URI from the CGI environment and returns a pair list which can be used by the server code to decide how to answer the request. The pair list keys are:

- `mode` one of `package`, `browse`, `upload` etc
- `package` package name
- `path` possibly empty, path inside package
- `query` query string
- `queryvals` query terms as a pairlist

`cantcl::parse_package_name` parses a package name into it's components, name, version and format. For example, `mypackage1.9b1.zip` is parsed as `{mypackage 1.9b1 zip}`.

Other procedures take care of setting up the CGI environment and performing the appropriate action based on the URI. The main CANTCL CGI script need only call `cantcl::cgimain` with no arguments to implement all server functionality.

The current server implementation keeps package archives as zip or starkit files in the filesystem and serves packages directly from this store, converting between formats as needed.

Since most of the work of the server is done with reference to the package metadata, this is stored in a Metakit database organised as an RDF triple store. Package descriptions are stored as triples of `{packageuri attribute value}` where `packageuri` is a unique identifier for the package generated by the system. The triple store is implemented by the `rdfstore` package and is designed to be compatible with a future Tcl RDF package. The triple store is updated either by asking the server to index an existing directory full of package archives or when a new archive is uploaded to the server.

When the server CGI script receives a request it first switches on the kind of request. For a browse or package request the triple store metakit database is opened to retrieve information about the package. A browse request will generate a result page as text, HTML or XML purely from the triple store. A package request will use the triple store to locate the appropriate file and then either return this file (perhaps after converting it to another format) or mount the file in order to serve one of the archive contents directly.

If CANTCL becomes a large volume application, the overhead of opening and closing the database file could be avoided by running it as a persistent server process via tclhttpd, mod_dtcl or Rivet.

5.2.1 Client Side Interface

Part of the `cantcl` package will be to provide a client side procedural interface to the CANTCL server. This will encapsulate the use of the appropriate URIs and so make it easy for client code to query the server for package details or download/upload packages.

5.3 Package `installer`

A long time ago, an `installer` package was the beginning of this project. Code was written to download and unpack archives from remote locations and provide various user interface elements for building installation tools. Very little work has been done on this recently and now that the server side tools are implemented this will be the major focus of the project.

The current `installer` package allows downloading and installation of packages. For example:

```
install_extension .../cantcl/soap.kit
```

This requires some kind of unzip facility, the easiest option being `zipvfs` but the code can fall back on an external unzip or prompt the user to unzip the file manually if this is not available.

Note that since the CANTCL server will allow the client to look inside a package archive, it might not be necessary to have a zip application on the client in order to install an extension.

A bootstrap module could been implemented to download and install, say tclvfs, in order to enable easier package installation in future.

In addition to downloading and installing packages there is a need for a client side interface to package browsing. In addition to allowing users to discover packages of interest via some GUI tool, this would allow the installer to locate package dependencies that were not satisfied in the current Tcl installation.

Another relatively simple client side tool would be to extend the package unknown handler to look for a package on CANTCL if it is not found locally. Since a canonical URI can be generated for any package (`http://purl.org/tcl/cantcl/package.version.format`) then it is trivial to query the CANTCL server for a package and call `install_extension` to make it available locally. One could even use `tclvfs` to mount the package directory over http in cases where no writable media are available.

6 Further Work

CANTCL is still a work in progress and this paper has presented only a snapshot of what is currently implemented and proposed. It is certainly the case that with actual use for a wide range of packages and client side situations, the interface described here will need to be modified. However, the server as it stands provides a workable package archive and the development of client side tools should show the benefits of this approach. There are of course a number of areas where it is already clear that work will be needed.

6.1 Starchive

Jean-Claude Wippler's Starchive [7] proposes a file store with very fine grained versioning of the individual files in a package. A particular file bundle is denoted by a unique version number which acts as a key via which the individual files can be located. Only one copy of each file version is stored in the Starchive. Hence, starchive provides a base versioned file store upon which the CANTCL URI interface could be implemented. The advantage would be reduced storage (no duplicate file versions) and more efficient access to files (from the MetaKit database rather than the file system). Once the CANTCL interface becomes accepted and client side tools are available we will pursue this server side optimisation.

6.2 Applications

CANTCL could make building custom package bundles and applications in starkit/starpack format much easier. One can imagine an application driven by a simple configuration file (possibly even using the TIP55 metadata format) which describes which packages are needed in an application or bundle. These could then be downloaded from CANTCL and assembled into a custom starkit for delivery. An alternative would be to offer this service on the CANTCL server.

I am currently using an extension of the TIP55 directory layout to store application scripts packages. The `apps` directory contains one or more application scripts which use the main package. It is relatively simple to turn a package structured this way into a starkit or starpack application.

6.3 Platform Dependencies

One area that has not been explored so far is the provision of packages containing shared libraries for a variety of platforms. In fact, while writing this paper it is clear that the proposed package download interface lacks a way of specifying the desired platform of the target package. This can become complicated if one allows multi-platform packages (which

TIP55 enables): how do I say "*please give me* `tclvfs` *for Windows-x86 and Darwin-ppc*"? One option would be to encode the platform in the package name (`foo_Linux-x86_1.3b2.zip`), another would be to use optional query terms appended to the package URI to specify additional constraints.

6.4 Authorisation and Security

Currently, file upload to the CANTCL server will accept any file for inclusion in the repository after a maintainer approves it. There is a clear need for various levels of authorisation for package contributors and for package users to be able to verify the integrity of a package archive. There are well understood ways of doing this via digital signatures but these will need some adaptation for CANTCL due to its ability to repackage archives in different formats and (potentially) to produce multi-package archives.

7 Summary

This paper has described the current state of the CANTCL project to build a canonical repository of Tcl packages. The proposal includes the specification of a web services interface to the repository allowing both browser based and automated clients to find and download packages. It is hoped that by the time of the conference, the CANTCL server will be running at `http://purl.org/tcl/cantcl/` providing the services described above. The main work remaining is to build client side tools for package installation and to populate CANTCL with as many packages as possible.

The CANTCL source code is available via the SourceForge project at `http://cantcl.sf.net` . Any and all input is welcome on the project.

References

[1] S. Cassidy. Ti55: Package format for tcl extensions. Tcl Improvement Proposal, August 2001. `http://purl.org/tcl/tip/55.html`.

[2] S. Cassidy. Defining a web services interface for a software package repository. In *AusWeb Conference*, July 2003. `http://www.ics.mq.edu.au/~cassidy/ausweb2003/`.

[3] Canonical R Archive Network. `http://cran.r-project.org/`.

[4] Debian linux. `http://www.debian.org/`.

[5] Dublin Core Metadata Element Set. `http://dublincore.org/documents/2003/06/02/dces/`.

[6] Tcl Extension Architecture. http://www.tcl.tk/doc/tea/.

[7] J.-C. Wippler. Starchive. `http://www.equi4.com/starchive`.