Jim and Tcl interoperability

Featuring *thing* o_o

Georg Lehner, MagmaSoft

EuroTcl 2023

Find this slideshow at https://vide.at/tcl/EuroTcl2023

Abstract

Whenever shell scripting does not fit anymore I use Tcl programs to handle system administration and automation tasks on my Linux servers.

Neither GUI nor speed is a requirement here, but rather frugal use of system resources. So I often build scripts or applications which run both with the Jim Tcl↗ interpreter and tclsh↗ .

This presentation:

- Shows some of the pitfalls encountered and how to circumvent them.
- Presents the framework I built to run the scripts seamlessly on both interpreters.
- Showcases some of the applications in daily use.
- Lists some interesting features encountered in Jim.
- Discusses *thing* : the yet unpublished minimalist interoperable o_o framework.

About

Me: Georg Lehner

- Personal homepage: https://jorge.at.anteris.net↗
- Company MagmaSoft e.U.: https://magma-soft.at↗
- Software Repository: https://at.magma-soft.at/darcs/↗
- Tcl Software: https://at.magma-soft.at/darcs/tcl/↗
- This slideshow: https://vide.at/tcl/EuroTcl2023↗
- I'm not a coder.

Coding Style and Context

- Variables are substantives, they are captialized like in German writing.
- Tcl version>= 8.6, but be conservative.
- Code for Linux/Unix, but don't introduce incompatibilities
- Tcl syntax is Tcl syntax: no ornamentation with extra quotes.
- Tcl has idioms.
- Take shortcuts.
- Documentation source code is POSH https://poshdoc.org↗ .
- Clean Code: *http://cleancoder.com↝*
- Take everything above with a grain of salt.

Why Jim

When running on machines with very low ressources, or on VM hosts with a huge number of VM's memory consumption is crucial.

```
What | tclsh | jimsh | diet jimsh


Size |  4.9M |  3.3M |  481k
VIRT | 18840 |  3936 |  960
RES  |  8764 |  2220 |  816
```

In Linux distributions, small shell implementations are used for scripts, bash is common for the CLI, e.g. in Debian:

```
What | bash | dash
VIRT | 9760 | 6124
RES  | 2576 |  936
```

Size of tclsh

tclsh8.6: 15k, libtcl8.6.so: 1.8M, libc.so.6: 1.9M, libz.so.1: 119K, libm.so.6: 887K, ld-linux-x86-64.so.2: 207K

My Whishes / Needs

for running Tcl scripts both with `jimsh` and `tclsh` :

- Same Tcl codebase.
  - At least almost the same.
  - Few if any `if {$tcl_platform(engine) eq "Jim"}` constructs.
- Create/reuse 'libraries'.
- Ad-hoc development and installation.
  - Distribution to third parties is less an issue.

Jim vs. `tclsh`

Considerations when running scripts both with Jim and `tclsh`

Tcl source in Jim:

- Only a subset of commands and libraries is available.
- Some commands behave differently.
- `package` 'ing works differently.

Jim specials:

- Dynamic namespaces.
- “Dynamic” arrays.
- `interp` 's are neither safe nor hierarchic.
- No variable tracing.
- Different string/utf-8 handling

Jim Commands with Behaviour

- `variable Name Value` .. not variadic.
- `file normalize` .. fails on non existing paths.
- `return` .. does not have `-options` .
- `socket` .. instead `socket` *`class`* , classes: `unix` , `stream` `dgram` ...
- `interp` .. misses `create` , `issafe` , `alias` .
- `info` ..
  - `info errorstack` .. not available, substitute: `info stacktrace`.
  - `info default` .. missing.
  - `info args` .. lists default values in sublist.

Other Differences in Jimsh

- `argv0` .. not always available, but `jimsh::argv` .
- `errorInfo` .. not available at all.
- `file normalize` .. fails on non existing paths.
- `~` .. is not expanded to users home directory.
- `encoding` .. missing.
- `binary encoding|decoding` .. missing.
- `trace` .. is missing, Jims' `xtrace` only traces `proc` 's.

Namespaces

Tcl: namespace is created on first usage.

Jim: namespace exists only during execution.

`namespace eval demo { variable Var }` ← Var vanishes in Jim.

`namespace eval demo { variable Var {} }` ← now it stays also in Jim.

Variable is named: `::demo::Var`

Excess colons are not supported: `::demo:::Var` fails

Arrays

If the value of a variable is a list with even length, then it is an `array` and a `dict` at the same time.

You can lappend to any variable.

`set Var(1) a` → `{1 a}`

`dict get $Var 1` → `a`

`lappend Var 2` → `{1 a 2}`

`array get Var` → `missing value to go with key`

`dict get $Var 1` → `missing value to go with key`

`array set Var {3 c}` → `missing value to go with key`

`array exists Var` → `0`

`lappend Var b` → `{1 a 2 b}`

`array exists Var` → `1`

Moral: You can kill your arrays with list operations.

So Why Jim and Tcl?

Technical

The subset of common syntax and behaviour is high enough to make it worth.

`tcltest` is available for both and runs from the same testfiles.

Conservative and sane programming leverages advantages of both environments

Political

Rubbing Tcl, Jim and Tcllib against each other improves each of them: bugs and fallacies tend to show up.

Make ressource-consciousness accessible to the community.

Personal

I understand Tcl better now.

It scratches my itch.

I sleep better with less bits burnt on my hardware.

Script Setup

A common, robust pattern for crafting scripts is needed for interoperability.

Let's use the name *demo* as a placeholder name for any script you want to create.

Rules

Every script *demo* is a Tcl package.

Every script `demo.tcl` resides in a directory named `demo`.

Install: symbolic link `demo` from a directory in *PATH* to `demo.tcl`.

Additional libraries are located in subdirectories named after themself.

Script Template

1. Shebang
2. Add the script directory to *auto_path* .
3. `package require` libraries.
4. `namespace eval` *demo* `{}` contains:
   - `variable Version` *x.y*
   - ... Author, Copyright, Description
   - `export` 's
   - Variables
5. `proc demo::main args {}` , then others,
6. `package provide demo $demo::Version`
7. if `source` 'd `return`
8. `demo::main {*}$argv`

The Shebang

Explicit

`#!/opt/diet/bin/jimsh`

Override the interpreter with: `tclsh $(which demo)`

Tentative

`#!/usr/bin/env jimsh`

Override like above.

Automatic Interpreter Selection

```
#!/bin/sh
# -*- mode: tcl -*- \
    hash -r; hash ${TCLSH:-jimsh} 2>/dev/null || hash tclsh 2>/dev/null; \
exec $(hash) "$0" ${1+"$@"}
```

Override the interpreter with: `TCLSH=tclsh demo`

Determine the script directory

The following commonly recomended method to find the real path with resolving the symlink on the last element breaks with Jim:

```
file normalize [file dirname [info script]/...]
```

Reason: `[info script]/...` is a path which on purpose does not exist. Tcl normalizes anyway, Jim throws error because of non existing file/directory.

Alternative:

```
if {[catch {file readlink [info script]} Path]} {set Path [info script]}
lappend ::auto_path [file normalize [file dirname $Path]]
```

…

```
if {![info exists argv0] || [file tail [info script]] ne [file tail $::argv0]} return
```

Libraries

Tcl: recursive search for `pkgIndex.tcl` in `$::auto_path` .

Jim: search for `demo.tcl` in `$::auto_path` . No version match is done.

Package *tcl4jim* provides `Tcl.tcl` which amends the Jim `package` command:

- Before the standard Jim search, `$auto_path` is searched for `demo/demo.tcl` .

Notes:

- Package name and filename must be the same .
- Package must reside in directory with package name.
- Required packages must reside in subdirectories.
- ... I said that before . :)

tcl4jim

A procject with some packages providing missing Tcl features for the Jim interpreter.

`Tcl.tcl` : add minimal emulation for often used missing features.

`mime.tcl` : copied from a 'recent' Tcllib and ported to Jim. Requires:

- `base64.tcl` : Pure Tcl, from Tcler's Wiki.
- `md5.tcl` : Pure Tcl, from Tcler's Wiki.

`http:tcl` : copied from http 1.0 and up'modded to http 2.0 interface.

Setup

Symlink `Tcl.tcl` to an `auto_path` directory, e.g. `/usr/local/lib/jim` .

`package require Tcl` – this also appends the *tcl4jim* directory to *`auto_path`* .

The Tcl package in *tcl4jim*

On top of `demo.tcl` script place:

```
package require Tcl 8.6
```

Emulations and improvements:

- `package` : added `provide` , `names` , `vsatisfies` .
  - `package require demo` .. also searches in demo subdirectory of each *`auto_path`* element.
- `interp` : added `create` , `issafe` , `alias` .
- `encoding` : added `names` .
- `socket` : reduce to work with `stream` only and use original syntax.
- `fconfigure` : add some options.
- `fcopy`
- `info` : added `args` , `default` , `errorstack` .

`tcllib` in Jim

Jim incompatibilities in tcllib, example `mime.tcl` :

- `return -options` .. used only once, can be avoided.
- `errorInfo` .. used a lot:
  - `catch {…} result; … set einfo $errorInfo; return … -errorinfo $einfo …`
  - becomes:
  - `catch {…} result einfo; … return -errorinfo $einfo …`

`mime.tcl` is still not working for all mime files, example:

- Under Jim: invalid header lines (utf-8 handling?).
- Under Tcl: `multipart/mixed` , loses parts.

Happens in DMARC rua messages (aggregate reports).

Some Jim/`tclsh` examples

- `logspook` / `spookview` : Log event tracking framework. Attack detection on IMAP/SMTP-Auth.
- `ipcheck/mmdb.tcl` : GeoIP lookup via external command. Spam classification.
- `tpdoc` : generate POSHDOC documentation from docstrings in Tcl modules.
- `rua` : process DMARC aggregated reports: https://brahe.magma-soft.at/dmarc/rua/prepacademytutors.com/2020/03/report.html↗
  - `mimedetach` : Best command line tool known to me for detaching attachment from RF2822 files.
  - `rfc2822/headers` : Header extractction for Spam classification.
  - `maildir` : Simplify operations on Maildirs↗ .
- `api` package: HTTP requests with Cookie jar.
  - `bbb_launcher` / `bbbtally` : BigBlueButton cluster swiss army knife. Uses `sxml` .
  - `CalTasks` : `caldav` package, requires `tdom` . WIP for Jim.
- `tsurbl` : Live Spam detection via embedded URLs in mail messages. Jim migration planned.

What about Object Orientation?

State of Affairs

Tcl: incr Tcl, TclOO, snit

Jim: pureTcl oo

None of them is interoperable with the other interpreter.

What would work for Me?

Simple objects are enough. Inheritance is not a topic.

Thingy: a one-liner OO system: https://wiki.tcl-lang.org/page/Thingy%3A+a+one%2Dliner+OO+system↗

Package *thing* o_o

~100 lines of Tcl code.

available at https://at.magma-soft.at/darcs/tcl/thing/↗ .

- *things* .. singleton objects.
- *makers* .. classes where each object is a variable a lá `string` , `dict` , ... in Tcl.
- *factories* .. “real” classes with methods.

No inheritance, (supposedly) fast method dispatching.

README and doc: https://at.magma-soft.at/darcs/tcl/thing/README.html↗ .

thing Things

```
package require Tcl 8.6
lappend auto_path .
package require thing
thing::thing Account
Account variable Balance 0
Account incr Balance 20
Account get Balance
Account proc show {{Chan stdout}} {
    puts $Chan "Balance of account is [get Balance]"
}
Account show
thing::destroy Account
```

thing Makers

```
thing::maker Account
Account proc init Self {variable $Self; set $Self 0}
Account proc init {Account {Value 0}} {
    variable $Account; set $Account $Value
}
Account proc show {Account {Chan stdout}} {
   puts $Chan "Balance of account $Account is [get $Account]"
}
set RichieRich [Account new 1000000]
$RichieRich show
Account incr $RichieRich 1000000
Account proc deposit {Account Value} {variable $Account; incr $Account $Value}
Account deposit $RichieRich 1000000
$RichieRich deposit 1000000
```

thing Factories

```
package require Tcl
lappend auto_path .
package require thing
thing::factory Account
Account proc init {Account {Value 0} {Type Debit}} {
    $Account variable Name $Account
    $Account variable Balance $Value
    $Account variable Type $Type
}
Account method show {Account {Chan stdout}} {
    puts $Chan "Balance of account [$Account get Name] is [$Account get Balance]"
}
Account method deposit {Account Value} {
  incr ${Account}::Balance $Value
}
Account create RichieRich 1000000
Account::RichieRich show
```

Thank You

Questions?

Credits

- Salvatore Sanfillipo: for creating Jim Tcl.
- Steve Bennet: for maintaining Jim Tcl.
- Felix von Leitner: for dietlibc.
- Richard Suchenwirth: for Thingy OO.
- The Tcl community: for sharing your knowledge.
- Rob Pike: for the Rio window manager, which inspired Console.
- Dan Bernstein: for showing how to program lean and portable. And for all of his software.
- Tim Berners Lee: for the Web.
- The Indieweb community: for microformats and the POSH acronym.

Making of 1/2

The slides are written in POSH - Plain Old Semantic HTML, with Neditor↗ and GNU Emacs↗ . They use only a small subset of HTML, carefully formatted with respect to line breaks. This allows to convert them to *THTMLN* with one `proc` consisting of three string manipulation commands. THTMLN – Tcl HTML Notation – is inspired by TON↗ .

Every slide has a `<nav>` element with one link to the previous slide, on to the first (the start) and on to the next slide.

The thtmln↗ package implements the HTML to THTHMLN converter and a THTMLN interpreter which occupies about 130 lines.

The slides are presented inside my Tcl/Tk Console↗ via the slide↗ package, which implements a frontend for the THTMLN interpreter, converting the slides to `text` widget formated text.

The slide package also includes a slide navigator and the utility `mkorder.tcl` to sort the slides according to the `<nav>` links.

init.tcl does all the loading, plus an adaptive scaling of the fonts according to the screen size of the presentation.

This setup allows me to execute any code sample given in the slides directly inside the console.

Making of 2/2

I was asked to create a single PDF from the slides.

To convert the slide HTML to PDF `wkhtmltopdf` is used. The non-patched version does this only file by file.

To get rid of the `<nav>` elements, which are superfluous in PDF, I recurred again to the `tdom` package, filtering out with another tool in the slide package: `preppdf.tcl` .

All pages are concatentated with `pfdunite` from the `poppler-utils` package.

Orchestration is done by redo-c↗ with two `.do` scripts: `default.pdf.do` and `slides.pdf.do` .

Jim Tcl on diet

dietlibc↗ is a minimal libc implementation for Linux.

```
./configure "CC=diet gcc -nostdinc -pipe" HAVE_PKG_CONFIG=0 --prefix=/opt/diet
make
```

yields a 481k static executable, including `zip` , but not `ssl` .

Standard glibc build without `ssl` : 3.3M

- jimsh: 396k
- libm: 887K
- libz: 119K
- libc: 1.9M
- Total: 3.3M