Tcl and C

Combining Tcl Scripting with C: The Best of Both Worlds
Tcl - History

- Tcl was created in 1988 by Dr. John Ousterhout at the University of California Berkeley.
  - He realized that large systems need some kind of scripting, or command language.
    - Examples: Microsoft uses VB, Emacs has lisp, many CAD programs use Tcl, web browsers have javascript, etc...
  - He also realized that it would be possible to make a simple, embeddable, reusable scripting language that could be easily integrated into applications
  - Ousterhout creates Tcl as a C library.
  - Tcl stands for “Tool Command Language” and is pronounced “tickle”.
Embedded Model

- The model originally proposed by Dr. Ousterhout was to have an application in C which could be extended in Tcl:
Embedding

- In this model, the application controls the flow of events, and occasionally runs a script.
- It's primarily useful when you use Tcl as a configuration language, or to handle certain events.
  - As an example, Apache + Rivet uses Tcl to create dynamic web pages.
- The main application is in control.
- Very useful way to add a lot of power to existing applications.
The second model relies on extensions in C being available as Tcl commands to the main program, which is written in Tcl:
Script first, ask questions later

- The idea behind the extension model is that you write the application first in the scripting language.
  - Then add pieces...
    - Special extensions
    - Optimize for speed
- Very powerful model, because development is very fast, but optimization is always possible.
  - There are many, many, many instances where a program was to be rewritten in C/C++ was left in Tcl (or Python/Ruby/whatever), because it worked quite well.
Like “Legos!”

It's easy to take blocks...
...and rearrange them!
Tcl Language

• Simple, and Quick.
  – Very simple syntax (everything is a command), and very flexible (you can write control structures in Tcl itself!). Tcl is “quick” - if you want fast, use C!

• Powerful, Lightweight, Adaptable
  – Tcl has lots of code available, and provides a lot of features for programming.

• Multiplatform
  – Runs on Unix, Windows, Mac (both OS X and older)

• Very liberal license
  – BSD license lets you embed it in proprietary applications
**Tcl Code**

- **Web programming**
  - Rivet, tclhttpd, AOLserver, http package, url parsing, cgi
- **Other networking**
  - dns, irc, ntp, nntp, pop, popd, smtp, multiplexer, snmp
- **File manipulation**
  - base64, csv, md5, mime, htmlparse
- **XML**
  - SOAP, tclxml, tclxslt, tcldom, tdom
- **GUI**
  - Tk, Gnocli
Everything Is A Command

• Really...

• Everything!

• if, while, foreach are not special like in C or Python
  – if { $var == 1 } { do something }

• This means you can create your own control constructs!
  – proc do {code while condition} ....
Simple web server

proc Serve {chan addr port} {
    fconfigure $chan -translation auto -buffering line
    set line [gets $chan]
    set path [file join . [string trimleft [lindex $line 1] /]]
    if { [catch {
        set fl [open $path]
    } err] } {
        puts $chan "HTTP/1.0 404 Not Found"
    } else {
        puts $chan "HTTP/1.0 200 OK"
        puts $chan "Content-Type: text/html"
        puts $chan ""
        puts $chan [read $fl]
        close $fl
    }
    close $chan
}

set sk [socket -server Serve 5151]
vwait forever
Overview of Tcl C API

- Variables
  - From C, to C, linked from Tcl to C, traces.
- Commands
  - Create commands.
- Interpreters
  - Safe interpreters
  - Create, destroy interpreters
  - Slave interpreters, share resources
- Threads
  - Basic thread operations – create/destroy/mutexes/shared memory.
- IO Channels
  - Create new channel drivers
  - Stack channels in order to create filters
  - Asynchronous
- Event loop
  - Create, destroy events
  - Schedule events
- Operating system interaction
  - Files
- Internationalization
  - Transform strings from and to different character sets
- Hash tables
  - Manipulate and use hash tables
Very Complete

- You can do almost everything you can do from Tcl, and in some cases more, from C.
- The API is well documented.
- In case you need more, Tcl itself is very well written source code, and is easy to understand, and modify.
- Tcl doesn't have object orientation built in... but the language is flexible enough that you can add OO as an extension!
C API Example

- Tcl does not have a built-in “kill” command to send signals to processes
- `int kill(pid_t pid, int sig);`
- In: PID to kill, signal to send
- Out: error on failure
Tcl / Other Languages

- We discuss Tcl, but the concepts are similar in other languages like Python, Ruby, Lua, etc...
- Interpreters
- Creating/Linking commands from scripting language to C
- Creating/passing variables and values between C and the scripting language
- Extending vs Embedding
Interpreter

- Interpreter is the “virtual machine” that processes a series of commands
- Tcl may have several interpreters active
- Tcl has safe interpreters
  - Run code in a sandbox
- Tcl code is byte compiled for fast execution
- Can be instantiated and controlled from both C and Tcl
- Can share resources
Tcl “Objects”

- Not “objects” like Java/Python/whatever
- Simply a unit that has:
  - A string representation
    - This must always be possible
  - Possibly, some other representation – int, double, long, string...
    - This is the more efficient form, usually
- In Tcl, objects use “reference counting” to do garbage collection
Create Commands

- `Tcl_CreateObjCommand(interp, "commandname", CfunctionName, clientData, deleteProc);`
  - Creates Tcl command that calls CfunctionName.
  - `clientData` is to pass extra data to the function.
  - `deleteProc` is called when the command is deleted.
Values

- `Tcl_GetStringFromObj(objPtr, lengthPtr);`
  - Gets string representation
    - Always returns something
- `Tcl_GetIntFromObj(interp, objPtr, intPtr);`
  - Gets int representation of object
    - Returns error if conversion is not possible
- `Tcl_NewStringObj(bytes, length);`
  - Returns new string object in Tcl
- `Tcl_NewIntObj(intValue);`
  - Returns new int object.
Variables

- Variables give Tcl a name to associate with a value.

- `Tcl_ObjSetVar2(interp, part1Ptr, part2Ptr, newValuePtr, flags);`
  - Sets variable name referenced by part1Ptr (and part2Ptr for arrays) to newValuePtr.
  - Equivalent to: set city “Trieste”

- `Tcl_ObjGetVar2(interp, part1Ptr, part2Ptr, flags)`
  - Returns the object that corresponds to the variable referenced by part1Ptr (and part2Ptr for arrays)
```c
#include <tcl.h>

int TclKill (ClientData clientData, Tcl_Interp *interp,
              int objc, Tcl_Obj *CONST objv[])
{
    int pid;
    int signal;

    if (objc != 3) {
        Tcl_WrongNumArgs (interp, 1, objv, "pid signal");
        return TCL_ERROR;
    }

    if (Tcl_GetIntFromObj (interp, objv[1], &pid) != TCL_OK ||
        Tcl_GetIntFromObj (interp, objv[2], &signal) != TCL_OK) {
        return TCL_ERROR;
    }

    if (kill (pid, signal) < 0) {
        Tcl_AppendResult (interp, "Error in kill: ",
                          Tcl_PosixError (interp),
                          NULL);
        return TCL_ERROR;
    }

    return TCL_OK;
}

int TclKill_Init(Tcl_Interp *interp)
{
    Tcl_CreateObjCommand (interp, "kill", TclKill, NULL,
                           (Tcl_CmdDeleteProc *)NULL);
    return TCL_OK;
}
```
Use kill command

> load ./tclkill.so
> kill 6471 1
... process killed ...
> kill 6471 1
Error in kill: no such process while evaluating {kill 6474 1}
Embedding Tcl

- Usually better to extend Tcl
- But if you have an existing application in C, you can make it much more flexible by adding a scripting language to it.
Tcl Init Functions

- `Tcl_FindExecutable("name of executable");`
  - Use `argv[0]`
  - Sets up initial bits of Tcl

- `interp = Tcl_CreateInterp();`
  - Creates Tcl interpreter. You can create as many as you like

- `Tcl_Init(interp);`
  - Initializes interpreter
```c
static void
MyApp_InitTcl(server_rec *s, pool *p)
{
    Tcl_Interp *interp;
    Tcl_FindExecutable(EXECNAME);
    interp = Tcl_CreateInterp();

    if (interp == NULL)
    {
        fprintf(stderr,
                "Error in Tcl_CreateInterp, aborting\n");
        exit(1);
    }
    if (Tcl_Init(interp) == TCL_ERROR)
    {
        fprintf(stderr, "Error in Tcl Init: %s\n", Tcl_GetStringResult(interp));
        exit(1);
    }
    Tcl_EvalFile(interp, "StartScript.tcl");
}
```

1) Interpreter is ready to evaluate files/scripts
Conclusion

• Script first...
• ...low level code later
• Create minimum necessary in C in order to favor code reuse