Tcl/Tk

Henning Schulzrinne
Dept. of Computer Science
Columbia University

Tcl/Tk

- C functions can become Tcl commands that are invoked interactively (cf. Unix executables → shell commands)
- Tk = scriptable, portable user interface
  - Windows, X (Unix), MacOS, MacOS X
  - also found in Python and Perl GUI extensions
  - scripts can send commands to each other (cf. DDE on Windows)

Tcl history

- Developed in late 1980s by John Ousterhout
- first release ~1991
- Tk usable around 1992
- see http://www.tcl.tk/doc/tclHistory.html

Tcl/Tk features

- high-level scripting language
  - less code than Motif or Win32
- interpreted
  - execute directly, without compiling or linking
- extensible
  - commands in Tcl or C
- embeddable
  - Tcl interpreter callable from C
- most platforms
  - Unix, Mac, Windows
  - hides UI, system call differences
- autoloading
  - automatically load libraries
- free
  - source
  - no royalties

Using Tcl/Tk

- Three modes:
  1. tclsh for interactive use
     $ tclsh
     % set x 7
     7
  2. wish for window programs
     $ wish
     % button .b –text "Hello" –command exit
     % pack .b

Using Tcl/Tk

- From C program:

```
#include <tcl.h>
main(int argc, char *argv[]) {
  Tcl_interp *interp = Tcl_CreateInterp();
  code = Tcl_EvalFile(interp, argv[1]);
  if (*interp->result != 0) {
    printf("%s\n", interp->result);
  }
}
```

Tcl/Tk script files

- Script file:

```
#!/usr/local/gnu/bin/wish -f
button .b -text "Hello, world!" \
 -command exit
pack .b
```


Tcl language structure

- Everything is a list of words – no fixed grammar
  - first word is command
- {} delay evaluation, may nest
- "" only needed when spaces:
  - `set x "foo"` = `set x foo`
- everything can be done dynamically: new procedures, variables, name spaces, ...
- interpreter model, but internally compiled into bytecode

Variables and substitutions

- Replacement like shell
- Substitutions:
  - variable substitution: `set a 17`
  - command substitution, evaluated as separate script:

    `set b [expr $a*4]`
  - backslash substitution: `set x \$a`

Tcl procedures

- procedures can be created dynamically

```
proc power {base p} {
  set result 1
  while {$p > 0} {
    set result [expr $result * $base]
    set p [expr $p-1]
  }
  return $result
}
```

- invoked as, say, `power 2 6`

Tcl event bindings

- binding: execute script whenever an event occurs (cf. handlers)
- e.g., `-command`
- more sophisticated: bind

Tcl binding example

```
#!/usr/bin/env wish -f
source power.tcl
entry .base -width 6 -relief sunken -textvariable base
label .label1 -text "to the power"
entry .power -width 6 -relief sunken -textvariable
  power
label .label2 -text "is"
label .result -textvariable result
pack .base .label1 .power .label2 .result -side left \
  -padx 1m -pady 2m
bind .base <Return> {set result [power $base $power];
  puts $result}
bind .power <Return> {set result [power $base $power]}
```

Tcl bindings

- widgets: labels, entries, buttons, ...
- `-textvariable` associates variable with display
- pack arranges the widgets into side-by-side, with spacing
- bind widget event Tcl-script, e.g.,
  - <Button-1> button 1 pressed
  - <a> key a pressed
  - <Motion> pointer motion

Tcl subprocesses

- unlike shell, commands are executed in same process
- but can 'exec' processes:
- `% exec ls`

Tcl – more details

- # comment
- one line at a time – use \ for continuation
- *"Tcl parses a command and makes substitutions in a single pass from left to right. Each character is scanned exactly once.*
- *At most a single layer of substitution occurs for each character; the result of one substitution is not scanned for further substitutions."*

Tcl – arrays

- array = collection of elements
- one dimensional only, but can simulate others
- but allow arbitrary subscripts → associative arrays

```
set earnings(February) 4827
set earnings($year,$month) 148
```

- `array` manipulates arrays:

```
array names earnings → January February ...
array size earnings → 12
```


Variables

- `incr` increments variable
- append adds strings to end:

```
append msg "more text"
```

- `argv` variable is list of command line arguments
- `env` is list of environment variables

Expressions

- Usually need '`expr`' command to evaluate, except in condition for if, while, ...

```
if {$x == $y} { ...}
set x [expr {$x + 7}]
set y [expr {log($y}]
```

- evaluates numerically where possible

Tcl lists

- list = ordered collection of elements
- separated by spaces or tabs
- any proper list can also be a Tcl command!
- concat *list list* – concatenate lists
  `concat {a b c} {d e f}` → `a b c d e f`
- join *list sep* – convert to string with separator
  `join {a b c} ", "` → `a, b, c`
- `lappend` *var element element*
  - append to end of list

Tcl list manipulation

- `lindex` *list index*
  `lindex {a b c} 1` → b
- `linsert` *list index value value*
  `linsert {a b c} 0 A {B C}` → A {B C} a b c
- `list` *value value*
  `list {a b c} {d e} f` → {a b c} {d e} f
- `llength` *list*
  `llength {a b c}` → 3
- `lrange` *list first last*
  `lrange {a b c} 1 end` → b c

Tcl lists

- `lreplace` *list first list value ...*
- `lsearch ?-glob|-regexp?` *list pattern*
- `lsort ?-command c? -increasing|decreasing` *list*
  `lsort -decreasing {a b c}` → `c b a`

Tcl control flow

- beware of line orientation – keep braces on same line as preceding:

```
if {$x < 0}
  { set x 0 }
```

- eval *arg ?arg arg ...?*
  - concatenate and evaluate

```
set x {expr 3+5}; eval $x
```


Tcl control flow

- `for` *init test reinit body*

```
for {set i 0} {$i < 10} {incr i} {puts $i}
```

- `foreach` *var list body*
- `if` *test1 body1* `elseif` *test2 body2 ...* `else` *bodyn*

```
if {$x < 0} {
} elseif {$x == 0} {
} else { ...}
```


Procedures

- variables are local unless declared
  `global x y`
- defaults:

```
proc inc {value {increment 1}} {
   expr $value + $increment
}
```

Procedures – call by reference

- upvar *?level? ovar myvar*

```
proc parray name {
  upvar $name a
  foreach el [lsort [array names a]] {
    puts "$el = $a($el)"
  }
}
```


String manipulation

- format *fmt ?value value ...?*
  `format "%2d" $x`
- scan *string format var ?var var ...?*
- string compare *string1 string2*
- string index *string charIndex*
- string length *string*
- string match *pattern string*
- string range *string first last*

regexp

regexp *exp string ?matchVar? ?sub? ?sub?*

.	any single character
^	matches null string at start of string
$	matches null string at end of string
\x	matches the character *x*
[c1-c2]	matches any single character
(regexp)	matches regexp – used for grouping
*	matches 0 or more of previous atom
+	matches 1 or more of previous atom
?	matches null string or previous atom
r1|r2	matches r1 or r2

regsub

- Substitutions based on regular expressions
  - `regsub there "They live there lives" their x`
- `regsub ?-all? ?-nocase? exp string subSpec varName`
- copies *string* to *varName*, substituting patterns in *exp* by *subSpec*
- & is replaced by matching substring
- *n* is replaced by *n*th () expression

regsub

```
regsub -all {(a+)(ba*)}
  aabaabxab {z\2} x
```

→ `2; x = zbaabxzb`

Tcl files

- typically, text files, but also binary files (→ binary, fconfigure)
- `open name ?access?`
  - returns `fileId` token
  - access: `r, r+, w, w+, a, a+`
  - can also read or write to pipe: `open |ls`
- close *fileId*
- gets *fileId ?varName?*
  - reads the next line from file

Tcl files

- read *–nonewline fileId*
  - read all remaining bytes in file
- read *fileId numBytes*
  - read at most numBytes
- puts *?-nonewline? ?fileId? string*
  - write string to stdout or file
- seek *fileId offset* ?start|current|end?
  - position within file

Tcl file properties – file command

file atime *name*	last access
file dirname *name*	directory name
file exists *name*	1 if file exists
file lstat *name array*	information about file
file size *name*	size of file
file tail *name*	name after last /
file type *name*	type of file
file writable *name*	1 if writable by user

File example

```
% file lstat mbox m
% parray m
m(atime) = 1017947827
m(ctime) = 1017947822
m(dev)   = 50856736
m(gid)   = 92
m(ino)   = 318887
m(mode)  = 33152
m(mtime) = 1017947821
m(nlink) = 1
m(size)  = 14636687
m(type)  = file
m(uid)   = 5815
```


Error handling

- Similar to C++ and Java
- catch {command} varName

```
if [catch {open foo.txt} msg] {
  puts $msg
}
```


time and date

- `clock seconds`
  - time in seconds (usu. since 1/1/70)
- `clock format`
  - convert to string
  - e.g., `clock format $t –format "%a, %B %e %Y %H:%M:%S"` → `Thu, April 4 2002 15:00:56`
- clock scan *dateString*
  - convert date string to integer

Tcl libraries

- `auto_mkindex` *`dir pattern`*
  - create tclIndex file to auto-load files
- `lappend auto_path dir`
  - add directory to path where Tcl looks for libraries
- `package require` *`name version`*
  - require a package; load if necessary
- `package provide` *`name version`*

Tcl internals

- `info exists varName`
  - returns 1 if variable exists, 0 otherwise
- `rename old new`
  - rename command old to new
- `trace variable name r|w|u command`

Tk overview

- widget = (X, Windows) window
- interacts with window manager (placement, decoration)
- application = single widget hierarchy
- widget have . names and are children of their parent widget (resizing, placement): .main.frame.zip
- . is topmost widget

Tk widgets

- Most .foo widgets are inside the toplevel window, but some can be toplevel themselves
- widgets can be created and deleted at run time
  - `button .b -text "Press me" -foreground red`
  - `destroy .b`

Geometry managers

- widgets don't determine their location or size on screen → geometry managers
- may depend on parent and sibling widgets
- widget only appears once given to geometry manager
- current geometry managers:
  - packer: sequentially around edges of cavity, with rows, columns
  - placer: fixed placements
  - grid: grid-like placement
  - canvas widget: position by coordinate
- can mix geometry managers in same application

Talking to widgets

- Widgets can be modified after creation
- automatically creates command named after widget

```
.b configure -foreground blue -text world
```

- `.b invoke` → invoke button as if pressed

Tk widgets: frames

- colored rectangular region, with 3D borders
- typically, containers for other widgets
- no response to mouse or keyboard

```
% foreach relief {raised sunken flat groove ridge} {
  frame .$relief -width 15m -height 10m -relief $relief -borderwidth 4
  pack .$relief -side left -padx 2m -pady 2m
}
% .flat configure -background blue
```

Tk widgets: toplevel

- Same as frames, except occupy top-level windows
- can indicate screen:

```
toplevel -screen displayhost:0.1
```


Tk widgets: label

```
% proc watch name {
  toplevel .watch
  label .watch.label -text "Value of
  $name: "
  label .watch.value -textvar $name
  pack .watch.label .watch.value -side
  left
}
% set country USA
USA
% watch country
```

Tk widgets: buttons, checkbuttons, radiobuttons

```
button .ok -text OK -command ok
button .apply -text Apply -command apply

frame .c
checkbutton .c.bold -text Bold -var bold -anchor w
checkbutton .c.italic -text Italic -var italic -anchor w
checkbutton .c.underline -text Underline -var underline -anchor w
pack .c.bold .c.italic .c.underline -side top -fill x

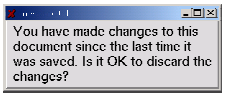
frame .f
radiobutton .times -text Times -variable font -value times -anchor w
radiobutton .helvetica -text Helvetica -var font -val helvetica \
   -anchor w
radiobutton .courier -text Courier -variable font -value courier \
   -anchor w
pack .times .helvetica .courier -side top -fill x -in .f
pack .ok .apply .c .f -side left
```

from Ousterhout

Tk widgets: messages

- like labels, but display multi-line strings

```
message .msg -width 8c -justify left \
 -relief raised -bd 2 \
 -font -Adobe-Helvetica-Medium-R-Normal--*-180-* \
 -text "You have made changes to this document
  since the last time it was saved. Is it OK to
  discard the changes?"
pack .msg
```



from Ousterhout

Tk widgets: listboxes

```
listbox .colors
pack .colors
set f [open /opt/CUCSX11R6/lib/X11/rgb.txt]
while {[gets $f line] >= 0} {
  .colors insert end [lrange $line 3 end]
}
close $f
bind .colors <Double-Button-1> {
  .colors configure -background [selection get]
}
```

from Ousterhout

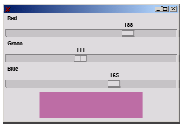
Tk widgets: scrollbars

```
listbox .files -relief raised \
  -borderwidth 2 \
  -yscroll ".scroll set"
pack .files -side left
scrollbar .scroll -command ".files yview"
pack .scroll -side right -fill y
foreach i [lsort [glob *]] {
  .files insert end $i
}
```

from Ousterhout

Tk widgets: scales

```
scale .red -label Red -from 0 -to 255 -length 10c \
  -orient horizontal -command newColor
scale .green -label Green -from 0 -to 255 -length 10c \
  -orient horizontal -command newColor
scale .blue -label Blue -from 0 -to 255 -length 10c \
  -orient horizontal -command newColor
frame .sample -height 1.5c -width 6c
pack .red .green .blue -side top
pack .sample -side bottom -pady 2m
proc newColor value {
  set color [format "#%02x%02x%02x" [.red get] [.green get]
   [.blue get]]
  .sample config -background $color
}
```



Tk widgets: getting values

- `-command`: e.g., scale invokes with new value, as in newColor 43
- `.widget get`: get value
- `-variable`: set variable
- event bindings

Tk widgets: entry

```
label .label -text "File name:"
entry .entry -width 20 -relief sunken -bd 2 -
   textvariable name
pack .label .entry -side left -padx 1m -pady 2m
```

Tk canvas

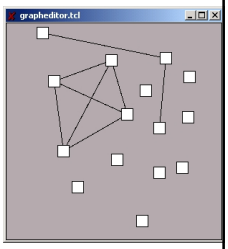
- display and manipulate graphical objects: rectangles, circles, lines, bitmaps, and text strings
- tagged objects → manipulate all objects with same tag (drag)
- event bindings for objects

Tk canvas example

```
canvas .c -width 12c -height 1.5c
pack .c
.c create line 1c 0.5 1c 1c 11c 1c 11c 0.5c
for {set i 0} {$i < 10} {incr i} {
  set x [expr $i+1]
  .c create line ${x}c 1c ${x}c 0.6c
  .c create line ${x}.25c 1c ${x}.25c 0.8c
  .c create line ${x}.5c 1c ${x}.5c 0.7c
  .c create line ${x}.75c 1c ${x}.75c 0.8c
  .c create text ${x}.15c .75c -text $i -anchor sw
}
```

Another canvas example

- canvas items generate names:
  - set mc [.c create circle ...]
- canvas items can be tagged:
  - .c create oval ... \ –tags myoval
  - .c delete myoval
  - .c itemconfigure circle –fill red
- several items can have the same tag
- one item can have multiple tags



The selection

- mechanism for passing information between widgets and applications
- first select, then get information about selection
- copy & paste, but also actions (DDD: set breakpoint)

Window managers

- each X display has a window manager
- controls arrangements of top-level windows on screen
- cf. geometry manager
- decorative frames
- iconify & deciconify
- examples: mwm, twm, fvwm95, KDE, Gnome, ...

Tk wm

- E.g., add title:
  - wm title . "Window Title"
- Iconify a toplevel window
  - wm iconify .w
- Normally, user cannot resize Tk windows, but
  - wm minsize .w 100 50
  - wm maxsize .w 400 150

Tk modal interactions

- Usually, user can select input focus (widget)
- modal interactions = restrict user choice
- example: dialog box forces user to fill it out before continuing
- `grab` restricts interaction to few windows
- tkwait suspends script until event
- use only in exceptional cases

Modal interaction example

```
button .panel.ok -text ok -command {
  set label OK
  destroy .panel
}
button .panel.cancel -text cancel -command
  {
  set label Cancel
  destroy .panel
}
pack .panel.ok -side left
pack .panel.cancel -side left
grab set .panel
tkwait window .panel
puts "label = $label"
```

Information about widgets

- winfo provides information about widgets:
  - winfo exists .w → 0 or 1
  - winfo children .w → .w.a .w.b
  - winfo class .w → Button

Tcl in C

- C implements objects
- manipulated by Tcl commands
- often, action oriented:
  robot turn r17
- object oriented: one command for each object (e.g., Tk widgets)

Tcl in C

- Two modes:
  - enhance wish or tclsh with additional C commands
    - use Tcl_AppInit()
  - add Tcl interpreter to existing C program
    - create interpreter

Example Tcl_AppInit

```
#include <tcl.h>
/* force inclusion of main from Tcl library */
extern int main();
int *tclDummyMainPtr = (int *)main;

int Cmd1(ClientData c, Tcl_Interp *interp, int argc, char *argv[]) {
  /* implement command here */
}

int Tcl_AppInit(Tcl_Interp *interp) {
  if (Tcl_Init(interp) == TCL_ERROR) {
    return TCL_ERROR;
  }
  Tcl_CreateCommand(interp, "cmd1", Cmd1, NULL, NULL);
  tcl_RcFileName = "/.myapprc";
  return TCL_OK;
}
```


Creating Tcl interpreters

- `Tcl_Interp *Tcl_CreateInterp(void)`
- `Tcl_Eval(Tcl_Interp *interp, char *script)`
- `Tcl_EvalFile(interp, char *fileName)`

Creating new Tcl commands

- `typedef int Tcl_CmdProc(ClientData clientData, Tcl_Interp *interp, int argc, char *argv[]);`
- `Tcl_CreateCommand(Tcl_Interp *interp, char *cmdName, Tcl_CmdProc *cmdProc, ClientData clientData, Tcl_CommandDeleteProc *deleteProc);`

Tcl C example

```
int EqCmd(clientData c, Tcl_Interp *interp, int
  argc, char *argv[]) {
  if (strcmp(argv[1], argv[2]) == 0) {
      interp->result = "1";
  } else {
      interp->result = "0";
   }
   return TCL_OK;
}
interp = Tcl_CreateInterp();
Tcl_CreateCommand(interp, "eq", EqCmd,
  (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
```

Tcl results

- ```
  typedef struct Tcl_Interp {
    char *result;
    Tcl_FreeProc *freeProc;
    int errorLine;
  }
  ```
- `interp->result` for constant strings
- Tcl_Result(interp, "string", TCL_STATIC);
- `TCL_VOLATILE`: on stack frame
- `TLC_DYNAMIC`: allocated via malloc

Tcl variables from C

- Tcl_SetVar(Tcl_Interp *interp, char *varName, char *newValue, int flags)
  - typically, global variable, but local if executed within function unless flags = TCL_GLOBAL_ONLY
  - Tcl_SetVar(interp, "a", "44", 0);
- char *Tcl_GetVar(Tcl_Interp *interp, char *varName, int flags)
  - value = Tcl_GetVar(interp, "a", 0);

Variable linking

- associate Tcl variable with C variable
- whenever Tcl variable is read, will read C variable
- writing Tcl variable → write C variable
- e.g.,

  int value = 32;

  Tcl_LinkVar(interp, "x", (char *)&value, TCL_LINK_INT);

Tcl references

- John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley
- http://www.scriptics.com/ has manual pages
  ```