DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing
March 15-17, Vienna, Austria
http://www.ci.tuwien.ac.at/Conferences/DSC-2001
K. Hornik & F. Leisch (eds.) *ISSN 1609-395X*

The R-Tcl/Tk interface

Peter Dalgaard[*]

Abstract

The `tcltk` package allows the use of the Tk graphical user interface elements from within R by embedding Tk commands into the R language. The design considerations are discussed, and also the shortcomings of the current implementation.

1 Introduction

It has been desirable for a long time to add graphical user interface (GUI) elements to R. This would be useful both to allow R programmers to write GUI-driven modules for specific purposes, and in the longer run to build a general GUI for entry-level R users.

It would be advantageous to build such features on a preexisting toolkit and several possibilities exist. In choosing between them, I put high priority on the possibility for rapid prototyping and the potential for multiplatform portability.

It was clear to me from the start that I would want a language embedding of the toolkit. It is not sufficient to build a predefined shell around R, it should be possible to write R[1] functions that modify the GUI, and perhaps even the entire GUI could be written in R. Preferably, it should be very easily to program, whereas efficiency in terms of speed may be less important.

Many cross-platform toolkits are in the form of C++ libraries or Java classes, but their object-oriented style seemed difficult to combine with the imperative programming style of R, although I have to concede that this could be attributed to shortcomings of my own.

[*]Department of Biostatistics, University of Copenhagen

[1]Much of the code could also work with other languages of the S family, but there is some R-specific use of environments

One toolkit which has been adopted by several other packages is Tcl/Tk. In all cases, it has led to considerable productivity, although perhaps not always complete satisfaction. Tk is designed to work with Tcl, a simplistic shell-like language and the target of much of the criticism. However, it has been successfully embedded in other languages, including Perl, Python (tkinter), and Scheme (STk, and very recently STklos).

The overall style of Tcl/Tk seemed to lend itself very easily to an R embedding and after a while it became quite irresistible to try and implement it, leading to the current `tcltk` package. As it turned out, a basic implementation could in fact be done very simply. The whole interface is only about 250 lines of C and 400 lines of R and about two thirds of the latter is repetitive function definitions.

2 Tcl/Tk

The following is a simple example of how Tcl/Tk works. Using the windowing shell (`wish`) you can create a simple button as follows (`%` is the `wish` prompt):

```
$ wish
% button .a
.a
% pack .a
% .a configure -text hello
```

The windowing shell creates a *toplevel widget*, called "." upon startup. This is used as a frame into which other widgets will be placed.

The Tcl language is very similar to a command shell. The basic form is that of a command followed by arguments and options. In the example, the first command creates a button widget ".a" within ".", but does not display it. The second command call upon a *geometry manager* (of which several are available) to position the widget within the toplevel widget. The third command is a *widget command* with the subcommand "configure" used to change the configuration of the widget – adding a label to the button in this case. This could also have been given as an option to the `button` command.

The most striking feature of Tcl/Tk code is its brevity. You do not need to specify every parameter which controls a widgets appearance; sensible defaults exist and the geometry managers save a lot of work by making explicit sizing of widgets almost unnecessary. Notice that the basic paradigm is one of scripting rather than class definition and instantiation as with traditional object-oriented languages. However, some notions from object-oriented programming do exist, for instance the fact that a widget might define its own methods is reflected in the notion of widget commands.

3 Embedding Tcl/Tk in R

3.1 The glue layer

It is not practically possible to bypass Tcl entirely and make R talk directly to the Tk widgets. It is necessary to go through Tcl, but it is possible to reduce Tcl to a thin glue layer, which users in practice won't need to know the details of.

It is very easy to set up communications between a C application and Tcl. Basically, once one has the Tcl event loop running, one can initialize one or several Tcl interpreters from the C code and start feeding them Tcl commands to evaluate. It is also possible to define new Tcl commands that when evaluated by the Tcl interpreter call C functions defined by the user.

The `tcltk` library sets up the event loop and initializes a Tcl interpreter with a couple of extra commands to handle callbacks to R. Then, as the most primitive communication method, a `.Tcl` function is defined to do little more than take a text string from R and pass it to the Tcl interpreter for execution.

3.2 Widget creation

Many aspects of Tcl commands map easily into R function calls, but the notion of a widget command causes trouble. In a widget command, the command *name* embodies the position of a widget in the widget hierarchy. With deeply nested widgets that can get quite unwieldy. An important difference between the two languages is that Tcl uses variable substitution like a shell script does. This helps making the widget command structure workable in larger programs, since it possible to store a prefix of the widget name in a variable, say `win` and have code like

```
text $win.txt
button $win.dismiss -text dismiss
pack $win.text $win.dismis
```

It is not attractive to mirror that mode of operation in R. For one thing, R does not easily lend itself to variable substitutions like the above, but the whole idea of having information embodied in a function name is alien to R – assigning a function to another name or passing it as a function argument does not normally change what the function does.

The same issue must have faced the designers of the `tkinter` interface to the Python programming language and they have solved it as follows:

```
root = Tk()
button = Button(root, text='hello')
button.pack()
```

Notice that apart from the assignment operator, the first two lines of Python might just as well have been R. In the third line the object-oriented nature of Python shows through, though.

In the above code, the `Button` command creates an object representing the button, based on a similar object representing its parent. The pathname of the resulting button window is not visible to the user, but stored somewhere internally.

In the `tcltk` package essentially the same idea is used, so that the following code in R is equivalent to the Python example above

```
root <- tktoplevel()
button <- tkbutton(root, text="hello")
tkpack(button)
```

(R does not have a namespace mechanism like Python does, so to avoid name conflicts all the public functions in the `tcltk` package carry a `tk` prefix.)

In the fundamentally object-oriented Python, the widget commands become methods for the widget objects. In R it is more natural to have functions acting on widgets, so that you might change the label of the button with

```
tkconfigure(button, text="goodbye")
```

There are a few cases where this leads to ambiguities, for instance does `bind` exist both as a regular command and as a subcommand for canvas widgets, to resolve this, the latter is implemented as `tkitembind`.

An object of class `tkwin` is implemented simply as a list with two components: `ID` holds the name of the window in Tcl and `env` is an R environment which holds information about subwindows, the parent window, and any callback functions associated with the window. The latter may appear slightly peculiar, but it ensures that if multiple copies of a `tkwin` object exist – this will happen when it is passed as a function argument for instance – all the copies will refer to the same environment, and the creation of (say) subwindows of one copy will be automatically reflected in the other copies as well. The need for recording this information in the first place is discussed in Section 4

3.3 Callbacks

There are two basic styles of callbacks in operation in Tcl/Tk. One results from scrolling commands and the other from event bindings. To link a listbox and a scrollbar in Tcl/Tk, one would typically use code like the following

```
.lb configure -yscrollcommand {.sc set}
.sc configure -command {.lb yview}
```

The values of the `-yscrollcommand` and`-command` options are *prefixes* of widget commands, i.e. further arguments will be tacked onto them when they are invoked. In the case of the `set` command for a scrollbar, the extra arguments will simply be two numbers giving the fraction of the trough where the two ends of the scrollbar slider are placed, but `yview` can be followed by three different styles of argument: `moveto x`, `scroll n units`, or `scroll n pages`.

The other style of callback is used when binding specific events as in (implementing simple wheel mouse bindings)

```
bind .lb <Button-4>  {.lb yview scroll -1 units }
bind .lb <Button-5>  {.lb yview scroll +1 units }
```

Here you give the entire script for Tcl to execute. However, you can also pass parameters into the script using *%-substitution*, as in

```
bind .canv <B2-Motion> {.canv scan dragto %x %y}
```

When the script is invoked, `%x %y` will be replaced by the mouse coordinates relative to the top left corner of `.canv`.

For the language embedding it is desirable, albeit slightly inefficient, to have callbacks that are R functions. The main problem with that is to communicate to the Tcl side that it should execute a given R function. Such a function might be unnamed and even when it is names, there are scoping problems that make it difficult to call the function from Tcl via an R expression. Instead, a small function (`.Tcl.callback`) exists which generates a script which invokes a dedicated `R_call` command taking the hex-encoded address of the function as the first argument. Simultaneously, the function is analyzed and if it has formal arguments apart from `...`; such arguments are converted to %-substitutions. For instance

```
> .Tcl.callback(function(x,y)cat(x,y,"\n"))
[1] "{ R_call 0x8369fdc %x %y }"
```

Obviously, only single-letter argument names make sense here. The `R_call` command calls the function specified by its first argument, passing any subsequent arguments on to the R functions.

With these definitions, it becomes possible to set up a scrollbar-controlled text widget with

```
txt <- tktext(tt)
scr <- tkscrollbar(tt,  command=function(...) tkyview(txt, ...))
tkconfigure(txt, yscrollcommand=function(...) tkset(scr, ...))
```

and one might also set up a binding which prints the mouse position when the left button is pressed with

```
blank<-tktoplevel()
tkbind(blank,"<Button-1>", function(x,y) cat(x,y,"\n"))
```

3.4 Option conversion

As seen from the examples already given, one can make a straightforward connection between Tcl commands and R function calls by letting argument names in R correspond to options in Tcl. The actual values can mostly be passed as text strings after conversion with `as.character` although some care has to be taken to escape characters that are special to Tcl (what Tcl programmers refer to as *quoting hell*), and also of course callback functions (see the preceding section) and `tkwin` objects need special treatment, the latter being converted to the value of their ID field. Some Tcl arguments are not given in option form but as keywords or subcommands without the leading '-' as `end` and `insert` in

```
.listbox insert end item1 item2 ...
```

Such items can be given simply as unnamed arguments. `NULL` is converted to an empty string, so in the few cases where you need to pass an argumentless option you can pass `name=NULL`. Vector arguments are automatically "flattened" by converting each element and pasting elements together separated by spaces. This allows you to do things like

```
tkinsert(listbox, "end", ls("package:base"))
```

All of this is handled by the function `.Tcl.args`, and there's a generic `tkcmd` function which is simply defined as

```
function (...)
.Tcl(.Tcl.args(...))
```

Almost all functions in the `tcltk` package are created as calls to `tkcmd`. The main exceptions are the commands that actually create widgets, since they need to create and return an object of class `tkwin`, and the code that handles the interface to Tcl variables.

3.5 Control variables

Several Tk widgets can be controlled by Tcl variables. For instance, a checkbutton can be set up and then turned of and off again with

```
checkbutton .check -variable foo
pack .check
set foo 1
set foo 0
```

Conversely, clicking the button with the mouse changes the value of `foo`.

This is something that it is not easy to map into R. The nicest thing to have would be if the link could be made to a specific R variable, but even though it is possible to link Tcl variables to particular memory locations (using the `Tcl_LinkVar` C function), R's variables do not occupy a permanent address. Instead we set up a pseudo list object, `tclvar` and overload the `$` and `$<-` operators for the corresponding class so that they call the Tcl `set` command. Thus the above example becomes

```
check <- tkcheckbutton(tt, variable="foo")
tkpack(check)
tclvar$foo <- 1
tclvar$foo <- 0
```

4 Garbage collection issues

When an object is no longer used by R, it can be removed by the garbage collector. However, when for instance a callback is passed to a widget, R will not automatically have a way of knowing that the function is still in use. This is handled by having a `.Alias` of each callback function stored in the window object to which it belongs, under a unique name. (The way this is achieved is quite underhanded since the relevant window could either be the result of a widget creation command or the argument of another command like `tkbind` or `tkconfigure`.)

However, saving a function with its window object is not too useful if the window object itself gets garbage collected. Hence each window object is stored in the environment of the parent window and the window itself keeps a variable containing the parent window. When a window is destroyed with `tkdestroy`, its entry in the environment of its parent is removed and it should thus be removable unless there is another link to it somewhere.

This system is far from perfect, but it does ensure that memory is not reclaimed prematurely, and with some care in programming, it should also allow memory to be reclaimed eventually.

5 Missing bits and problems

The `tcltk` package is still in a somewhat experimental state. There are several things that do not quite work as one might desire, although users have been quite productive with the current setup.

Somewhat to my surprise, performance in terms of speed seems to have been a minor issue. For instance, the inefficiency of having extra parsing layers between a text widget and its scrollbar is hardly noticeable. However, it is possible to stress the system beyond its capability – for example starting the `tkpager` on a large file can take a long time. There are also some cases where the Windows version runs very slowly, but these are likely due to shortcomings of the event loop handling.

In the following subsections, I will sketch some of the problems that will have to be addressed in the near future.

5.1 Return values

Currently, essentially nothing is done about return values from Tcl commands. They are simply returned in string form, even when the value being asked for is known to be numeric. This puts the burden of conversion on the programmer. For instance, it does not work to use

```
if (tkwinfo("exists", tt)) {...
```

because the string return value cannot be interpreted as a logical value. It can be handled reasonably cleanly by using

```
if (tkwinfo("exists", tt) == 1) {...
```

but the necessity for this sort of coding would be better avoided.

However, whereas it is simple to convert nearly anything to string form, as done by `.Tcl.args`, conversion in the opposite direction is harder. In fact, it is impossible without knowledge of what type the value is supposed to be of; a value of “1.0” might after all be a string and distinct from “1.00”. It would be possible to write code to deduce the return value from the call that was made, but it is quite a daunting task. A better idea is probably to use “Tcl objects” as obtained from `Tcl_GetObjResult`. The common non-string types (boolean, integer, double) should be deducible from a `Tcl_Obj`.

5.2 Globality of control variables

As discussed, the `tclvar$` method of accessing control variables is less than ideal. Binding directly to R variables is not possible without changes to R itself, though. However, there is another problem associated with control variables, namely the fact that such variables are global in Tcl. This means that if one runs two instances of (say) the `tkttest` demo, buttons checked in one of them also get set in the other. In Tcl itself, one can make such variables unique by attaching a prefix to the name, which is possible but unattractive in R. Perhaps some way of encoding the local scope is what is needed here.

5.3 Embedding graphics

Currently the `tcltk` package has no way to get R graphics into a Tk widget. One can use the Tk canvas widget, or generate the usual graphics windows, but not for instance add a Tk menubar to the R graphics. There are several approaches to obtaining such functionality and Luke Tierney has done some preliminary work, including a method where an off-screen bitmap is generated and placed in a widget. Alternative methods include window reparenting techniques (which are non-portable) and writing a dedicated device driver for the Tk canvas (which needs some trickery to allow rotated text).

5.4 Event loop handling

The current method for running the Tcl event loop pseudo-concurrently with R and its graphics devices are quite crude. I strongly suspect that a better understanding is needed of the synchronization issues involved and that it would be well worth studying the “notifier” structures that Tcl sets up when running pure Tcl/Tk applications.

5.5 Documentation

The documentation for most of the commands in the `tcltk` is virtually absent. However, since the rules for converting R commands to Tcl are quite transparent, one can mostly get by with using the documentation for Tcl/Tk.

6 Perspectives

The main purpose of developing the `tcltk` package was to enhance R with GUI elements. The focus of development is currently on getting the details of the interface right, but it might be a good idea to take a little time to look at what might be achieved.

The most obvious perspective is to develop a menu and forms based interface to common statistical procedures. Such interfaces exist for many other statistical systems and even though they invariably fall short in terms of flexibility compared to language-based approaches, they do make life considerably easier for entry-level users.

For expert users, menus tend to be an obstacle rather than a help. However, GUI elements could still be useful for them. In my experience, it is often simple things that are most valuable, e.g. obtaining a list of the variables in a data frame while setting up a model formula. If it can be obtained by the click of a mouse, so much the better.

In teaching with R, it can be difficult to achieve a smooth transition from command line usage to function writing. A simple method for writing, editing and executing short scripts would be useful, and this kind of functionality can be obtained quite neatly using text widgets.

I have mentioned the problems associated with getting R graphics into a Tk widget. However, one might also note that there are things that can be done with the Tk canvas that are not easily obtainable with R, such as moving or deleting graphics objects and groups thereof. The possibility of using at least some of these features as inspiration for developing a completely new graphics model in intriguing.

As a general comment, I have found Tcl/Tk quite pleasurable to work with, in particular because its basic spirit is quite similar to R (and S). It does have its shortcomings, but none that appears insurmountable, and it is still under active development and supported by quite a large body of users and developers.

7 Web resources

`http://www.tclfaq.wservice.com/tcl-faq/`
`http://kaolin.unice.fr/STk/STk.html`
`http://www.perltk.org/contrib/ptkFAQ.html`
`http://www.pythonware.com/library/tkinter/introduction/`