Klant – A Server-Managed Client Framework for Tcl/Tk

Steve Landers
Digital Smarties
steve@DigitalSmarties.com

Abstract

The server-managed client model promises the best of both worlds – the richness of a thick-client user experience with the convenience and security of the thin-client model. This paper describes Klant – a prototype Tcl/Tk server-managed client framework that builds upon several existing Tcl technologies to show what can be achieved using this amazing language.

Background

The IT purchasing criteria for many organsations is "Fit for Purpose, Value for Money". In the never ending search for productivity improvements, management has concentrated on the latter and sought to drive down costs by deploying web based and thin client applications. But often such applications don't provide the rich user experience of more traditional thick client application. Put another way – you can do so much more with Tk than you can with HTML.

On the other hand, thick client applications have their own issues – in particular the higher cost (to the user, developer and administrator) of deployment, maintenance and management of such applications.

The ideal solution would combine the flexibility and quality of a thick client solution with the lower costs of thin client – achieving both "fit for purpose" and "value for money".

In May 2004, IBM announced a middleware strategy called "server-managed clients" that sought to bring the together the benefits of web-based computing and traditional desktop applications[1].

But there are a number of Tcl-related technologies that have facilitated the development of centrally managed, easy to deploy, shared data applications for some time

- Tk is cross-platform and textual – and so can be distributed at run-time[2]
- Starkits provide a simple deployment model[3]
- TLS – an SSL extension for Tcl providing secure client/server communication[4]
- TclRFB provides a scriptable client side implementation of the VNC (Virtual Network Computing) protocol[5]
- Tequila enables arrays to be shared amongst clients with data stored centrally[6]

In the spirit of "never let a new buzzword pass you by", these have been combined into a framework called Klant – that provides a prototype for building Tcl/Tk server-managed client applications.

Overview

The Klant server runs on Unix systems with a VNC[7] server installed. The client will run on just about any system that supports Tcl/Tk and TLS – including Linux, Windows and MacOSX.

Klant is deployed as two Starkits – one for the server side, and one for the client side. The intention is for these Starkits to be customised for each application.

The server Starkit listens for and accepts connections from clients, authenticates them, starts a VNC server and then invokes the application. The listener runs as root (so that it can authenticate the user and invoke the application under the correct User ID) but once started the server side of a client/server pair runs as the authenticated user. Only the listener process stays running as root.

The client includes a VNC viewer, implemented using the TclRFB extension. This allows server applications to be displayed on a client without an X11 server or VNC viewer installed. But the client also includes a framework for incrementally adding functionality in the client that can be controlled from the server side applications.

The client includes a simple embedded web server – so that HTML documents can be stored in the client and displayed locally on the client computer using a browser.

Although the Klant server is deployed as a single starkit, it contains code for a number of processes and forks multiple times to create each of these processes. There is a single listener process that accepts connections from clients and instantiates the server side of a client/server pairing, comprising a number of processes

- a server process that authenticates the user and then securely tunnels I/O to and from the client
- an Xvnc process that runs the VNC server
- an application process that runs the application code (whatever that might be)

At its simplest, Klant can be used as a drop-in self-contained VNC client – for example, to display a Unix/Linux based application to Windows desktops without needing to install an X client or VNC viewer. The application displays output like any other X11 application – output is sent to the VNC server (indicated by the DISPLAY variable), which passes it to the Klant server, which in turn tunnels it back to the Klant client where it is displayed using TclRFB.

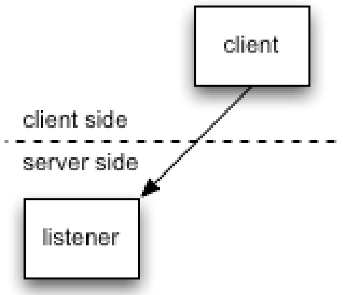
But it also has another path back to the client via the server, which can be used by the application developer to incrementally add functionality on the client side that can be invoked from the application. This can either be functionality within the main Klant window, or external programs such as browsers or word processing software.

For example, one prototype built for a customer used this to invoke additional functionality locally on the users desktop (either Windows or Linux), including

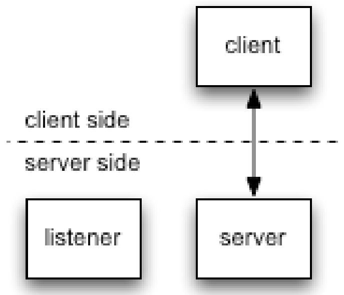
- application specific features stored in the client Starkit
- OpenOffice for editing documents that are stored centrally on the server
- invoking a browser to view documentation stored within the client Starkit's virtual file system (VFS), using the simple web server built into the client

Architecture

When the server starts, it listens for connections on a specified port. The listening socket is opened using TLS – so that it supports SSL connections from a client.

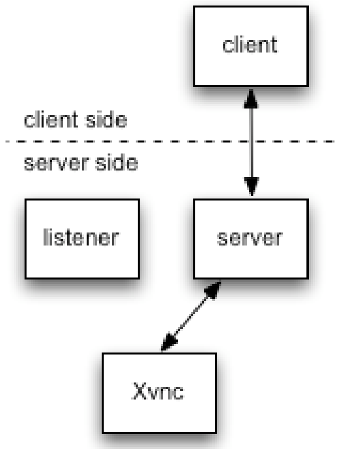


When a client starts, it displays a login screen that allows the user to specify a user name, password and the server to connect to. The client connects to the listening port on the specified server, using TLS to ensure the connection is secure.

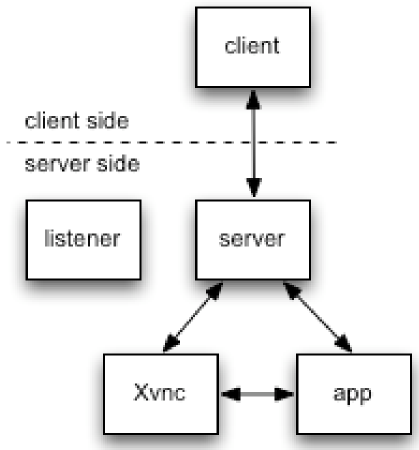


On accepting a connection, the listener forks a child process to run as the server side of the client/server pair. The server process inherits the secure bi-directional socket connection back to the client, and signals to the client that it is ready to read commands by writing a start up message to the socket.

Once it recognises that the server is ready, the client sends the user name and password to the server, which attempts to authenticate them using the PAM (Pluggable Authentication Module) facility[8]



Following a successful user authentication, the server forks another process to run a VNC (i.e. Xvnc) server. This process is connected back to the server process by a bi-directional socket, which is connected to standard input and standard output. Xvnc itself is started using the –inetd flag so that instead of listening for TCP connections it uses its standard input and standard output. The server process tunnels output from the Xvnc process back to the client, which has been waiting for Xvnc to start.



The server then creates another listening port and forks another process to run the application. Before executing the application start up command a few environment variables are set

- DISPLAY – the X11 display number
- CLIENTPORT – the server TCP listening port
- LOGNAME – the user name

The intention of the listening port is to allow applications to connect back to the client (tunnelled via the server) to invoke additional application specific functionality in the client.

The end result is a triangular relationship, with X11 output going to the client via Xvnc and the server, and a separate path from the application to the client (again via the server) to invoke added functionality in the client.

After starting the Xvnc and application processes, the server sets its User ID and Group ID back to that of the authenticated user – so all three server side processes run with normal user privileges.

While this has been happening, the client has been waiting for Xvnc to start (remember that Xvnc output goes to stdout, which is tunnelled back through the server to the client). Once it detects Xvnc has started, the client initialises the TclRFB viewer code and displays the application.

Any application code to be run in the client must be stored in the client Starkit. The application should open a socket to the localhost listening port specified in the CLIENTPORT environment variable. This is actually the Klant server, which will pass anything written to the socket through the secure tunnel back to the client (and vice versa).

It is up to the application developer as to how this channel is used, but a simple line buffered text API that invokes procedures in the client works well. At it simplest, this would mean a big switch command in the client that invokes specific procedures and returns the resulting values.

Implementation

Klant is implemented in [incr Tcl] [9]. Although it is a relatively small application (the server is approximately 350 LOC and the client is around 630 LOC) it is the author's contention that the encapsulation and abstraction provided by an Object Oriented approach is preferable when building graphical applications.

The question could be asked as to why [incr Tcl] and not one of the alternatives? In particular, Snit[10] could have been used but there were concerns about performance – especially considering Klant needs to be deployed with Tcl/Tk 8.4 (which does not have the ensemble support that promises to improve Snit performance). Alternative OO extensions weren't considered. Whilst it might have been an opportunity to use XoTcl[11] or one of the other more modern ones, until such time as an OO system is "blessed" by the TCT it was judged more appropriate to use [incr Tcl] from the start given the author's familiarity with it, and that it is included in both Tclkit and ActiveTcl.

The TLS extension is used to provide secure communications channels between the client and server. TLS is an OpenSSL/RSA-bsafe Tcl extension that provides secure connections on top of the Tcl socket mechanism. It was originally developed by Matt Newman, but is now maintained by a team via a SourceForge project.

Configuring TLS is relatively straight forward, and socket operations are analogous to the standard Tcl equivalents, for example

```
::tls::init -certfile cert/server-public.pem \
            -keyfile  cert/server-private.pem \
            -ssl3 1 -require 0 -request 0
set listen [::tls::socket \
                  -server [list $this accept] $listen_port]
```

Authentication is performed by the Klant server using PAM - the Unix/Linux Pluggable Authentication Modules facility. A new Critcl[11] extension was built to enable access to PAM from Tcl scripts. Invoking this in the server is simple …

```
if [pam::authenticate $user $pwd] {
     # authenticated
} else {
     # authentication failed
}
```

The TclX[12] extension is used in the server for access to the Unix fork and execl commands.

Invoking fork and execl is simple enough, as shown by this code fragment that starts the VNC server.

```
set listen [socket -server [list $this vnc_accept] 0]
if {[fork]} {
     setusergroup $user
     set port [lindex [fconfigure $listen -sockname] 2]
     close $listen
     set parent [socket localhost $port]
     close stdin
     close stdout
     set in [dup $parent]
     set out [dup $parent]
     set err [dup $parent]
     fconfigure $in -blocking 0 -buffering none \
                    -translation binary
     fconfigure $out -blocking 0 -buffering none
                    -translation binary

     set args [list -inetd –ac -query localhost -once \
                    -geometry ${width}x${height} \
                    –depth $depth]

     catch {execl /usr/X11R6/bin/Xvnc $args}
     # fall back
     catch {execl /usr/bin/Xvnc $args}
     # give up
     puts stderr "couldn't exec Xvnc"
     exit
}
vwait VNCready
```

As you can see, Tcl plus TclX is a perfect choice for scripting Unix/Linux system administration. Whilst Perl seems to have that space tied up, it could be argued that Tcl is just as useful, if not more when considering the added advantages of event driven I/O and the Tk toolkit.

Another Critcl extension was built to provide access to Unix user and group library functions

- set/get user ID
- set/get effective user ID
- set/get group ID

For example, here is a code fragment that shows the Critcl definition of the setusergroup procedure that is used by the Klant server to lower its privileges after starting Xvnc and the application process …

```
package require critcl
package provide uid 1.0

critcl::ccode {
    #include <sys/types.h>
    #include <unistd.h>
    #include <pwd.h>
}

critcl::cproc setusergroup {char* name} int {
    struct passwd *pwd = getpwnam(name);
    if (pwd == NULL) {
            return 0;
    }
    initgroups(name,pwd->pw_gid);
    setgid(pwd->pw_gid);
    setuid(pwd->pw_uid);
    return 1;
}
```

Then this is called in the server thus ...

```
package require uid
…
setusergroup $user
```

Mac Cody’s TclRFB extension is used to implement the VNC viewer in the client. TclRFB is a pure-Tcl implementation of the VNC Remote Framebuffer (RFB) remote desktop facility. It supports version 3.3 of the RFB protocol, and provides both client and server facilities (although only the client is used in Klant). Multiple RFB encodings are supported - Raw, CopyRect, RRE, CoRRE, and Hextile. Sample VNC client server applications are provided.

A minor change was made to TclRFB to enable it to work with TLS – a new procedure ::rfb::ImportClientSocket was added to enable TclRFB to use a pre-existing socket. This allows the Klant client to create a socket connection with the server using TLS and then instruct TclRFB to use that socket.

The TclRFB sample client was used as the basis for the VNC viewer code, but converted to an [incr Tcl] class.

The Klant client’s embedded web server is based on Jean-Claude Wippler’s hpeek[13] code, and provides a simple web server that serves pages stored within the client Starkit’s html directory.

The Klant client provides a skeletal method that implements a simple API protocol for the application to invoke functionality in the client

```
method command {sock} {
    if [eof $sock] {
        exit    ;# client has died?
    }
    set code 0
    set result “”
    if {[gets $sock line] ne “”} {
         busy::hold .
        if {[catch {set result [run_command $line]} msg]} {
            set code 0
        } else {
            set code 1
        }
        busy::release .
    }
    return [list $code $result]
}
```

This accepts a single line from the application, runs it (using the run_command method described below) and returns a single line to the client. This single line is a list comprising two elements – the first is the return code from the command, the second is any output.

Note the use of the busy::hold and busy::release commands. These are the busy widget from the BLT[14] toolset and have been extracted into a stand-alone package using Critcl and included in the Klant client Starkit. The busy widget is used to lock the user out of parts of a GUI application whilst performing some other activity.

The run_command method should define the actual functionality supported in the client on behalf of the application. For example, the following defines a couple of useful commands

- invoking OpenOffice[15]
- display a document from within the Starkit using an external browser

The run_command implementation could also call application specific functionality implemented within the client in Tcl/Tk.

```
method run_command {line} {
    set cmd [lindex $line 0]
    set args [lrange $line 1 end]
    switch $cmd {
    OpenOffice {
        if ::tcl_platform(platform) eq "windows"} {
            set cmd [file nativename \
  "c:/program files/openoffice.org1.1.1/program/soffice.exe"]
        } else {
            set cmd [auto_execok soffice]
        }
    }
    document {
        set url http://localhost:$http_port/$args
        if {$browser eq ""} {get_browser }
        switch $::tcl_platform(platform) {
            unix {
                exec $browser $url &
            }
            windows {
                if {$::tcl_platform(os) eq "Windows NT"} {
                   exec $::env(COMSPEC) /c start $url &
                } else {
                    # Windows 95/98
                   exec start $url
                }
            }
        }
    }
}
```

The get_browser command just looks for an available browser on *nix

```
method get_browser {} {
    set browser ""
    switch $tcl_platform(os) {
      Linux {
        foreach cmd {firefox mozilla konqueror galeon \
                                        netscape} {
            if {[lindex [auto_execok $browser] 0] ne ""} {
                set browser $cmd
                break
            }
        }
      }
      Darwin  {
          set browser open
      }
   }
}
```

Issues

Deploying TLS and certificates proved troublesome. Partly this was due to the author's unfamiliarity with TLS, programming OpenSSL[16] and, in particular, the whole question of certificate management (and even whether they are necessary). Ideally, it should be possible to make the Klant server only accept connections from a client with a valid certificate. Given that Klant is a prototype, this is no big deal at present – but input from someone more familiar with the topic would no doubt help.

Another issue with TLS is that certificates need to be outside the Starkit since the underlying OpenSSL library isn't VFS aware. Probably the simplest way of addressing this would be to copy the certificate out of the Starkit to a temporary file before initialising TLS, and then deleting it afterwards.

The other issue is that TLS is statically linked (for security reasons). But this leads the server Starkit being sensitive to the Linux platform it is being run on due to different versions of glibc. There's probably no solution apart from making multiple versions of the server Starkit.

There are a number of issues with Xvnc.

Firstly, the behaviour of different Xvnc versions seems to vary, particularly when using the –inetd flag. Klant was developed and tested using RealVNC, but in time it should be possible to find a combination of flags that enable it to be run with other versions providing they support version 3.3 of the VNC RFB protocol.

Another Xvnc issue is that of authentication. When the Klant server starts Xvnc it doesn't yet know the X11 DISPLAY that will be used. This is obtained by scanning through the temporary files created by Xvnc - there is a file /tmp/.X*N*-lock for each – where *N* is the display number of the particular Xvnc instance. Inside this file is the process number of the Xvnc, which is compared with the process number of the Xvnc process created by the Klant server to find which value of DISPLAY to set in the application process.

```
proc getdisplay {pid} {
  # get next display name by poking around in /tmp
  foreach lock [glob {/tmp/.X[0-9]*-lock}] {
    set fd [open $lock]
    gets $fd p
    if {$p == $pid} {
      regexp {/tmp/.X([0-9]*)-lock} $lock xx display
      return $display
    }
  }
  return ""
}
```

Whilst less than elegant, this works fine in practice except for one issue. The Klant server doesn't know which DISPLAY will be assigned, and so can't set up the authentication tokens using the X11 xauth command. Ideally, xauth would be used to instruct Xvnc only to accept connections from the local host, using a private cookie that Klant generates and passes to both the Xvnc server and to the application.

The solution used was to disable authentication using the Xvnc –ac flag. This isn't ideal, and so other solutions are being considered – including attempting to force Xvnc to use a specific display.

Klant performance also needs addressing. Whilst sufficient for proof of concept, the performance is nowhere near as fast as a native VNC viewer such as that provided for Linux and Windows. Some tuning of the VNC encodings used has helped, but Klant hasn't yet been profiled to identify the bottlenecks.

And the final issue is that debugging is horrendous. It's not just that there are three processes on the server side, but also the behaviour of Xvnc changes depending on if it is connected to a socket or a terminal. Practically speaking, the only debug tool is "puts"[1] used in conjunction with sockspy to see what Xvnc is sending.

Opportunities

There is potential to improve Klant's performance. Profiling should reveal which of the following are contributing to the slower performance compared with standard VNC

- TclRFB performance (it is, after all, pure Tcl code)
- TLS performance, including the choice of algorithms and key sizes
- the tunnelling overhead - i.e. the server copying I/O from Xvnc to the client
- [incr Tcl] procedure and method dispatching overhead

If TclRFB performance needs addressing, then Critcl could be used to speed up parts of the RFB code. This would mean compiled versions for common platforms (presumably Linux, OSX and Windows), but still having a fallback Tcl implementation for other platforms.

Consideration could also be given to using zlib to compress the VNC protocol to reduce bandwidth (although this isn't likely to be a significant factor since standard VNC seems to work just fine).

There is currently no mechanism for storing application code in the server uploading (or is it downloading?) it to the client at run-time – either on start up or as needed. Currently such code needs to be stored in the client Starkit. This should be relatively simple to implement (perhaps via the Tcl Virtual File System) and would be a good reason to implement the zlib support on the application/client communications channel.

In addition, there is potential to build Tequila support into the clients so that any such application code could share arrays with other clients, although this will require another shared process - perhaps the listener, or maybe a process that controls access to a database such as Metakit[17] or SQLite[18].

[1] It is acknowledged that some consider this the only necessary debugging tool in all situations ☺

And finally, there is the potential to integrate and embed Internet Explorer into the Klant client on Windows, using techniques similar to those used in Michael Jacobson's NewzPoint application[19]. NewzPoint uses OpTcl[20] to embed Microsoft's Internet Explorer ActiveX component in a Tk frame. A similar technique might be feasible on Unix/Linux using an X11 based browser (eg. Mozilla or one of its derivatives) and George Peter Staplin's TkXext[21] extension. This would allow web-based applications to be deployed using an application specific customised client.

Conclusion

Klant started as a fun project, somewhat inspired by Mac Cody's comment on the TclRFB web page – "the intent of TclRFB is to enable the creation of light-weight, scripted remote interfaces" – but also in response to a comment from Jean-Claude Wippler about the IBM's Server-Managed Client announcement ("It dawned on me that we've been sitting on way too many things without using it to its best").

But it has grown into a useful demonstration of just how flexible and powerful Tcl and associated technologies are. Klant goes considerably further than usual in terms of gluing and integration by delivering a fully standard X11 mechanism across diverse desktop platforms, plus tying into the desktop applications such as browsers and office products like OpenOffice.

It has already been incorporated into a customer prototype, where it was used to deploy a Linux based application to Windows desktops, but with added functionality running locally on the desktop.

The code is available under a Tcl friendly BSD/MIT license – basically, free for any legal use (including in proprietary products) as long as attribution is given. For use without attribution a commercial license is available.

References

[1] *IBM throws weight behind server-managed clients*, The Register, May 13th 2004 http://www.theregister.co.uk/2004/05/13/ibm_server_managed_clients/
[2] The Tclers Wiki Tk page - http://wiki.tcl.tk/tk
[3] Starkits – http://www.equi4.com/starkit.html
[4] TLS – http://sourceforge.net/projects/tls
[5] TclRFB - http://tclrfb.sourceforge.net/
[6] Tequila - http://www.equi4.com/tequila.html
[7] VNC – Virtual Network Computing - http://www.realvnc.com/
[8] PAM – Pluggable Authentication Modules for Linux - http://www.kernel.org/pub/linux/libs/pam/index.html
[9] [incr Tcl] - http://incrtcl.sourceforge.net/itcl/
[10] Snit - http://www.wjduquette.com/snit/
[11] Critcl - http://www.equi4.com/critcl.html
[12] TclX - http://sourceforge.net/projects/tclx/
[13] hpeek - *Inspecting app state with a browser* - http://mini.net/tcl/9154
[14] BLT - http://blt.sourceforge.net/
[15] OpenOffice.org - http://www.openoffice.org

[16] OpenSSL - http://www.openssl.org/
[17] Metakit - http://www.equi4.com/metakit.html
[18] SQLite - http://www.sqlite.org
[19] NewzPoint - http://www.tcl.tk/newzpoint
[20] OpTcl - http://www2.cmp.uea.ac.uk/~fuzz/optcl/default.html
[21] TkXext – http://www.tcl.tk/tkxext