Proxy Tk: A Java applet user interface toolkit for Tcl

Mark Roseman
TeamWave Software Ltd.
roseman@teamwave.com

ABSTRACT

Proxy Tk allows a Tcl program to provide a highly interactive web-browser user interface, without requiring the end user to download additional software. It uses a thin client design, where a server-side Tcl application communicates with a very small generic Java applet running in the browser, sending it commands to create and modify widgets, and receiving events from the user. A Tk-like layer encapsulates the communication protocols to provide a familiar programming paradigm, and allow easy porting of existing code.

Introduction

For some applications, running in a user's web browser rather than as a conventional workstation application is important. Existing Tcl solutions support building a wide range of web-based applications, but do not address interactive user interfaces in situations where downloading extra software may be problematic.

This paper describes Proxy Tk, an architecture that allows very interactive user interfaces to be developed in Tcl and run within a web browser, without the need for the end user to download additional software.

It uses a thin client design, where the main Tcl application, rather than running on the user's workstation, runs on the server. It directs a small Java applet running in the user's web browser to provide its user interface. Because this applet is generic, it supports a range of different applications, and can be extended for specific cases. This approach leverages the high-level, rapid development, and easy extension capabilities of Tcl, while still delivering a web user interface through the Java applet.

We begin the paper by examining the demand for web based software, using our own TeamWave Workplace application as an example. After considering several alternative Tcl development approaches, we provide an overview of our solution, and then proceed to detail its two key features, the communication protocols and the Tcl interface. We then examine how we used Proxy Tk to migrate our own software from a standalone to a web based application, and discuss issues that can arise using this approach in various other applications.

Though not a panacea, Proxy Tk solves an important part of the problem of delivering web applications using Tcl.

Web-Based Software

For many application areas, there is a strong preference for software that is delivered to users via their web browser, rather than via a traditional application installed on their own workstation. Arguments made for web-based software are varied, grounded both in reality and perception.

For example, one-time or sporadic tasks may not justify the time, effort, resources and security checks needed to install new software. Downloading new applications or installing system add-ons may be too complicated for novice users [4]. Many people (wrongly!) argue since users already know how to use browsers, web-based applications will have no learning curve.

These various concerns often end up encoded in organizational policies, making it even more difficult to adopt software that is not web-based.

While HTML or Javascript solutions are alternatives for many form based applications, programs requiring greater richness or interactivity must rely on either Java or a browser plugin approach. Unfortunately, we hear many of the same sentiments about downloading new applications expressed about plugins (except of course those bundled with the browsers).

TeamWave Workplace

As one example of the demand for web-based software, consider our own TeamWave Workplace application. One or more users running a client application connect to a server, where they share data and communicate with each other, using chat, whiteboards, and other tools. The user interface is shown in Figure 1.

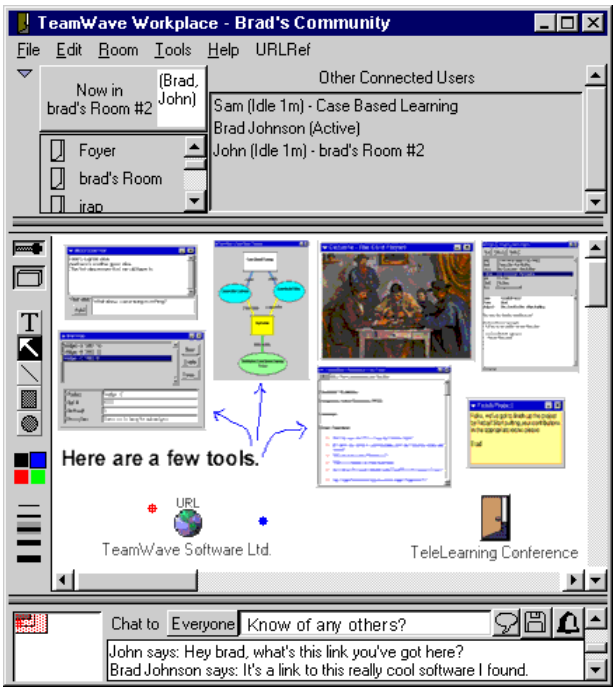

Figure 1. TeamWave Workplace user interface.

Both the client and server are large programs, written in a mix of C/C++ and Tcl/Tk. The client is particularly complex, containing several different Tcl interpreters, one for each of the tools. The architecture of an earlier prototype of the system is described in [6].

While many users were fine with downloading and installing our client (completely “wrapped” and a simple install), we continued to receive requests for a web version. Groups running “virtual communities” needed to lower the barriers for new users to join and to use the community frequently. In practice, users would not make the extra effort to download the software unless they had a very pressing need. In some situations (e.g. Internet cafes, labs), installing software to use the community was impractical. For some of our key markets, a web-browser version was essential.

A Java version seemed the right thing to do, but would involve throwing away our considerable code base. We had also just released a Tcl SDK for developers to extend the application, and could see no way to allow this from the Java side.

Finally, we were not thrilled by the prospect of trying to develop a large Java applet, when so many who had attempted this had failed. We needed something that would preserve our code base as well as the robustness and rapid development we were accustomed to with Tcl, yet still run as a Java applet.

Existing Approaches

Tcl developers have several excellent options to them for developing web-based applications, each suited to different types of applications.

Microscripting. Using tclHttpd or another Tcl-savvy web server, small Tcl scripts can be executed on the fly and their results embedded in the HTML page [7]. This solution is restricted by what is possible to deliver in HTML, and therefore cannot be used to deliver a sufficiently rich and interactive user interface.

CGI. Using a library such as cgi.tcl, running against any web server, Tcl developers can easily write CGI scripts to do a variety of forms processing [3]. Again, the user interface presented by CGI scripts is limited to what can be provided by HTML.

Jacl. By providing an implementation of Tcl written entirely in Java, this approach has promise for delivering more interactive user interfaces [1]. However, the current version is both incomplete and cannot run as an applet within a web browser.

VNC. This program [5] (and similar software) takes a Unix X11 desktop and sends it to a remote viewer, similar to how the X11 protocol can work over a network. While a Java applet viewer is available, this solution works with only the entire desktop, not a single window. Multiple instances of the application cannot be run off the same machine; multiple users all share the same desktop display. Further, because it operates at a very low level, high bandwidth and low latency networks are essential.

Tcl/Tk Plugin. The Sun Tcl/Tk Plugin allows running full Tcl/Tk scripts safely in a web browser [2]. Based on the standard Tcl/Tk code base, the plugin is capable of very sophisticated user interfaces. However, because the plugin is not bundled as a standard part of existing browsers, it requires an extra download and install step, which as we have seen may be an obstacle for some applications. Even if the plugin were bundled with existing browsers, a custom plugin would be needed for applications requiring any C extensions.

In this paper, we consider a new approach, capable of delivering very interactive web-based interfaces, yet not requiring users to download additional software.

Architecture Overview

We are basing our approach on a “thin client” design. In a conventional application architecture, the code for the application runs entirely on the user’s workstation, using the workstation’s own windowing system to present its user interface. This is known as a “thick client” design.

In the “thin client” design we are presenting here, rather than a single component running on the user’s workstation, the system consists of two software components, the “thick” server and the “thin” client. The server is a Tcl program running on a network server, and implements the bulk of the application. This is the *same* code that would normally run on the user’s workstation in a thick client design — we are now migrating this code to instead run on a network server. This Tcl application communicates with a thin client, which is a generic Java applet, running in the end user’s web browser. This thin client provides the user interface for the application.

Though our application is inherently network oriented, in any situation where web browser solutions are called for, servers must exist, making this architecture viable. We’ll consider in our case study later in the paper how to apply this architecture to applications that in their original form use a client-server architecture, where the main application running on the user’s workstation talks to a server program on a remote machine. But again, in the normal case, we’re talking about web-enabling applications that used to run entirely on the user’s workstation, with no server component.

The applet doesn’t know about our specific application, but is able to respond to simple commands sent from the Tcl server program. These commands direct the applet to create its user interface, and to communicate user interactions back to the server. The Java applet then consists solely of a general purpose "widget server," used by the server-side Tcl program to build the complete user interface at run-time.

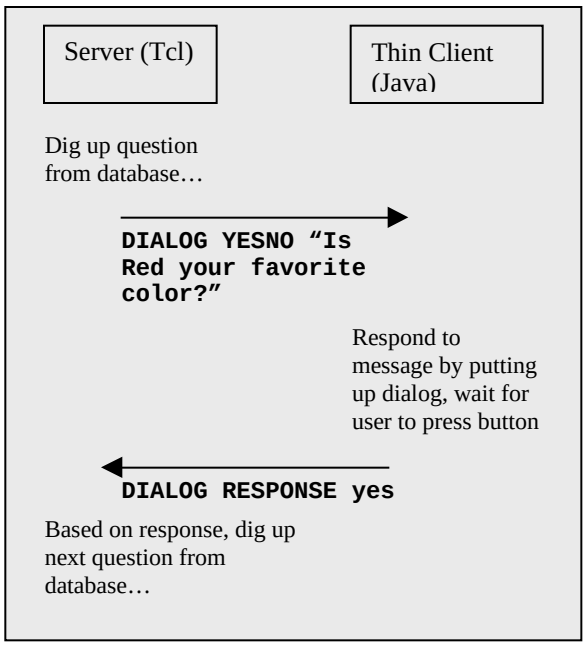

Figure 2. Example of thin client interaction.

As an illustration, consider an application that asks a series of questions of the user, the next question based on the answer to previous ones. Using our architecture, the Tcl server application will know about the different questions and the logic to chose between them. All our Java thin client needs to know is how to ask a question of a user and send the response back to the server. A simple exchange is illustrated in Figure 2.

This dialog example is still fairly high level; the thin client needs considerable knowledge about how to ask a question of the user, even if it doesn’t know about the specific questions to ask. What we’ll need to make the thin client more general is to work at a lower level, more akin to individual Tk widgets.

Consider what could be built if your Tcl program could communicate to the Java thin client using something like a Tk canvas widget. It could instruct the applet to create various line, shapes, text and other items. The thin client could respond to mouse events by sending messages back to your Tcl program, which you handle in a way specific to your application. It is at this level of granularity that we’ve actually constructed this architecture.

Design Goals

When designing this system, we had several goals in mind, both reflecting our particular requirements and also what we perceived as the potentially wide-ranging uses of this thin client approach.

Run in current browsers. Because we actually wanted to deploy this software, we needed to balance our desire to use the latest and greatest Java features in the applet against what people were actually running. We chose to target browsers supporting JDK 1.1.

Simplify the thin client. We wanted to design the system, including the network protocol, to make the Java thin client side as simple as possible. Keeping less code in the applet not only makes for a faster download, but allows us to leverage C code extensions on the server for performance and more functionality.

Don’t assume Java. Where possible, we wanted to design the protocol so that it was more generic, and didn’t assume Java on the thin client end. This would let us consider other implementations, such as e.g. ActiveX on Windows, or design thin clients for platforms that may not support Java well, or at all, e.g. PalmPilot. As such, not only should the protocol syntax not rely on Java-specific tricks (e.g. serialized objects), but the message contents should minimize assumptions about how the Java client would be implemented.

Easy Tcl interface. We didn't want developers to have to interact with communication protocols directly, at least not unless they were adding new capabilities to the toolkit (analogous to coding a new widget in C). The natural choice is a Tk-like interface, where a "proxy widget" on the server interacts with the thin client using the communication protocols. In keeping with the goal of simplifying the thin client, advanced features (e.g. canvas tags) would normally be implemented strictly in these server-side proxies; the thin client would know only about object ids.

Extensible. The protocol should be extensible, so that new functionality can be added if required for particular applications (e.g. more features, better performance). The Tcl server application should also be able to determine if extensions are present on the thin client, and to take advantage of them if so.

Communication Protocols

This section describes the lower level communications protocols upon which the Proxy Tk architecture has been built. We'll describe the actual communications mechanisms by stepping through an application of how it would be actually used.

In our example, we are running our Tcl server application on the machine www.company.com. This machine is also running a web server; because of Java security restrictions, our applet will only be able to connect to servers on the same machine as it is retrieving web pages from.

Making Connections

A user on another machine uses their web browser to open up the web page at this URL:

http://www.company.com/myapp.html

This page contains the following HTML:

```
<html>
<body>
<applet code="ThinClient.class"
        width="400"
        height="400">
<param  name="host"
        value="www.company.com">
<param  name="port"
        value="9900">
</applet>
</body>
</html>
```

When this web page is loaded, it will run the applet stored in ThinClient.class, passing as parameters the host and port of our Tcl server application. On starting, the applet will make a socket connection to this server.

The Tcl server application is written something like the following. It sets up a listening socket, and waits for connections from a thin client, When it gets one, the server creates a new Tcl interpreter for that particular thin client. It then loads a Tcl package called "proxy" into the new interpreter, which provides routines to send and receive messages from the thin client, defines proxy widgets, etc. It then transfers the socket to the new interpreter. Finally, it sources the Tcl program containing the code for our application. That program will then communicate with the thin client.

Using multiple interpreters, one for each client, removes the possibilities for namespace conflicts between client proxies, and also opens up the possibility of using one thread per thin client, for performance. Though not a fundamental characteristic of our architecture, using multiple interpreters did prove to be a useful design choice in our case.

```
# Main server application code
socket -server acceptCmd 9900

proc acceptCmd {sock addr port} {
   set interp [interp create]
   interp transfer "" $sock $interp
   $interp eval package require proxy
   $interp eval proxy::init $sock \
      $addr $port
   $interp eval source myapp.tcl
}
```

Messages

Both the Tcl server-side and the Java thin client watch their sockets for *messages* sent from the other side. Though any reasonable framing strategy could be used to define the boundaries of messages, we chose simple carriage return delimited strings, with appropriate quoting for carriage returns embedded in the message.

The actual format of the messages is not defined, other than it must start with a single word (i.e. sequence of non-space characters), and may optionally be followed by a space and any further sequence of characters.

This first word signifies the *message handler* that is responsible for handling this command. Both the Java side and the Tcl side consist of a number of these handlers, which determine the functionality they are able to provide (i.e. what messages from the other side they respond to). A variety of built-in handlers are included when you call "proxy::init" on the Tcl server, or when you instantiate the ThinClient applet from Java. You can also add others.

Handlers have a name (which is the first word of the network messages), a version (for tracking enhancements over time), and a handler proc which

responds to the message. On the Tcl side, this is just a Tcl procedure taking a single argument, namely the extra parameters sent along as part of the network message. A handler proc is registered using the "proxy::handler" Tcl command e.g.

```
# handler for FOO version 0.1
proc fooHandler {params} {
  ... interpret params
}

proxy::handler FOO fooHandler 0.1
```

Server side handlers can also be defined in C; a similar API is used in that case. A C handler proc consists of a single function taking both a string parameter (for parameters), and a pointer to an internal structure holding data for that proxy client.

On the Java side, a handler is any class that implements an interface called MessageHandler, containing a single handle method. To add new packages to the applet, the programmer generally creates a new subclass of the ThinClient class, and calls its registerHandler method to add each new package. This is illustrated below.

```
class FooHandler
       implements MessageHandler {
  public void handle(String params) {
    ... interpret params
  }
}

public class MyClientApplet
     extends ThinClient {
  public void registerHandlers() {
    super.registerHandlers();
    m_client.registerHandler("FOO",
       "0.1", new FooHandler());
  }
}
```

Determining Capabilities

One of the requirements in this system is that the application logic contained in the server needs to know what packages the thin client implements, so that it knows what facilities for display are available to it. This might be used for example, so that different applets (with more features) can be run on newer browsers that support more recent versions of the JDK.

When the thin client first connects up, it sends a message to the server telling it what handlers (and what versions of each handler) it has available. The message might look something like this:

```
HANDLERS CANVAS 1.0 ENTRY 0.5...
```

On the Tcl side, the first word of the command is stripped out ("HANDLERS") and the remainder of the command sent to the handler for the (built-in) handler named HANDLERS. This handler stores the list, and makes it available to the Tcl application via the "proxy::remotehandlers" command. With no parameters, this command returns a list of all known handler names, while passing it the name of a handler returns the version of that particular handler.

The message format above would be fairly typical for most handlers, because it is easy to parse. But again, it is worth stressing that this system assumes nothing about the message format after the first word. The routines that read in messages just look for the first word in their list of registered handlers, and send the rest of the message on to the appropriate handler.

Defining Widgets

Not surprisingly, each widget that is provided by the thin client implements a single handler. For example, the thin client's button widget is accessed through messages starting with the word "BUTTON". Most widgets have some elements in common, such as an id number to refer to them, a fairly uniform way of formatting the underlying messages (e.g. "handler id operation options…") and similar operations that are available (e.g. create, delete, set configuration options).

To use the button widget as an example, the server would send the following two messages to the thin client to create a button (having id 25) and set its label:

```
BUTTON 25 new
BUTTON 25 set label Push Me!
```

On the Java side, a singleton instance of a ButtonHandler class would receive and interpret the messages. The new message would cause the handler to create a new instance of a ButtonWidget (a wrapper around AWT's Button class) and store it in a table of all known widgets, indexed by its id. The set message would cause the handler to look up the ButtonWidget in the table, and call its 'set' method, which would modify the properties of the underlying AWT button.

Notice that the Java side has not received any instructions as to what to do when the user presses the button (e.g. the equivalent of the "-command" option in Tk). The thin client has no knowledge of how the application should deal with events; anytime the button is pressed, it will simply send a message back to the server, which can handle it as it chooses:

```
BUTTON 25 buttonPress
```

The application code running in the server will have to remember that button 25 corresponds to the "Push Me!" button, and know what to do when it is pressed. While it would be possible to develop programs at this low level that use the Java thin client, clearly a higher level interface that hides all these protocol details would be beneficial. We'll cover just such a programming interface in the next section.

Tcl Interface

The previous section outlined the communications protocol which defines the low-level interface between the server-side application and the Java thin client display. The specific operations supported by the protocol determine what functionality in the thin client is available to the application for its use.

However, its clear that working at such a low level — formatting and parsing network messages, caching proxy widget state information, and keeping track of widgets by id number — are not exactly conducive to the high level programming practices we've become used to with tools like Tcl and Tk!

Therefore, we created a Tcl interface that encapsulates all the low level protocols, caches widget information, and does all the housekeeping one would expect.

Not surprisingly, the design of this Tcl interface bears more than a passing similarity to Tk. A widget creation command (e.g. button) is used to create new widgets, their object command (e.g. .proxy.b) is used to refer to them and manipulate them, configuration options can be set and queried, event bindings are available, etc. Table 1 provides a summary of what subset of Tk has been implemented in the current version of Proxy Tk.

Because the API is mostly just a subset of the API for Tk, it's a fairly straightforward manner to either port existing code, or have a code base that runs well under both Tk and Proxy Tk. We'll discuss this in a bit more detail in the next section. We'll also touch on some of the differences that arise because the actual user interface is running remotely from the application, and how these can be addressed.

The implementation of the proxy widgets on the Tcl side is also straightforward. Each widget typically requires three things. A message handler interprets messages it receives from the thin client. A widget creation command creates both a new Tcl command to refer to the widget, and a cache for any data associated with the widget, Finally, a widget object command is used to set configuration options and perform other operations. Each of these will format and send messages over the network to the thin client.

Command	Options
button	-text, -command, -font
entry	get, insert, -width, -show, -font
menu	post, add/insert; command, separator, radiobutton, entryconfigure; –label –command –state
text	insert, delete, get, -width –height –font –color –state –background
listbox	insert, delete, get, see, curselection,–font
canvas	create line, rectangle, oval, text, window, image; bbox, canvasx, canvasy, delete, coords, itemconfigure, itemcget, bind, find withtag / overlapping / enclosed, type, gettags, raise, lower, –width, –height, –text, –background, –scrollregion; –tags, –fill, –outline,–font, –anchor, –window
grid	–in, –column, –row, –columnspan, –rowspan, –sticky
bind	which events depends on widget
destroy	
focus	-force
winfo	width, height, exists

Table 1. Summary of current Proxy Tk Tcl interface.

Example

To give a quick example of the correspondence between the Tcl interface and the underlying network protocol, consider the freehand drawing program shown in Figure 3. Buttons at the top control the color

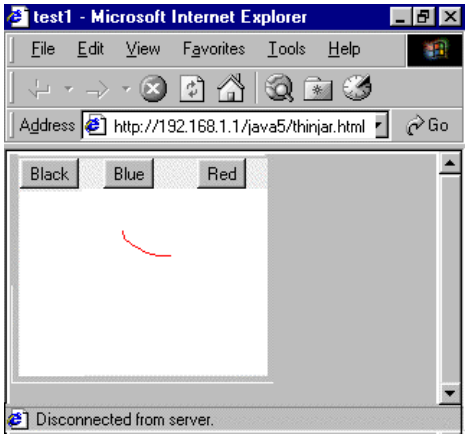

Figure 3. Simple Proxy Tk drawing program.

of the drawing. Clicking and dragging in the canvas draw in the selected color. Code is below.

```
# simple freehand drawing program

# root window
set w .proxy

# import commands into the root namespace
namespace import proxy::*

# create buttons at the top for choosing
grid [button $w.black –text Black \
   -command “set color black”] \
   –column 0 –row 0
grid [button $w.blue –text Blue \
   -command “set color blue”] \
   –column 1 –row 0
grid [button $w.red –text Red \
   -command “set color red”] \
   –column 2 –row 0
set color black

# canvas for drawing
grid [canvas $w.c –background white] \
   –column 0 –columnspan 3 –row 1 \
   -sticky nwes
bind $w.c <1> “set x %x; set y %y”
bind $w.c <B1-Motion> {
   $w.c create line $x $y %x %y \
      –fill $color
   set x %x; set y %y
}
```

When this program first starts up, the following network messages are sent from the thin client to create the user interface (comments in italics).

```
# create each button; the grid is handled
# by the root window (id=1), which is a
# canvas widget in Proxy Tk
BUTTON 2 new
BUTTON 2 set text Black
CANVAS 1 grid activate
CANVAS 1 grid add 2 column=0 row=0
  columnspan=1 rowspan=1 fill=none
  anchor=center

BUTTON 3 new
BUTTON 3 set text Blue
CANVAS 1 grid add 3 column=1 row=0
  columnspan=1 rowspan=1 fill=none
  anchor=center

BUTTON 4 new
BUTTON 4 set text Red
CANVAS 1 grid add 4 column=2 row=0
  columnspan=1 rowspan=1 fill=none
  anchor=center

CANVAS 5 new
CANVAS 5 set background ffffff
CANVAS 1 grid add 5 column=0 row=1
  columnspan=3 rowspan=2 fill=nwes
  anchor=center
```

If the user clicks the red color button, the following message is sent back to the server.

```
BUTTON 4 buttonPress
```

Finally, as the user clicks and draws on the canvas, messages like the following sequence are sent from the thin client, followed by a message back from the server, telling the client to actually draw the line on the display.

```
# mouse down and drag event
CANVAS 5 mousePressed x=26 y=32
CANVAS 5 mouseDragged x=30 y=34

# server sends create a line item, id 6
CANVAS 5 create line 6 26 32 30 34
CANVAS 5 itemset 6 color ff0000

# event from client…
CANVAS 5 mouseDragged x=32 y=36

# … and server tells us to draw, etc.
CANVAS 5 create line 7 30 34 32 36
CANVAS 5 itemset 7 color ff0000
```

Proxy Tk in Practice

We used Proxy Tk to port our TeamWave Workplace software described earlier to run in a web browser. In this section, we’ll describe our experiences with doing the port, and identify some of the issues that arose as a result of using this thin client architecture, and the solutions we found.

In describing how we used Proxy Tk, we can’t claim to provide a perfectly objective evaluation. The Proxy Tk infrastructure was developed in conjunction with the port of Workplace, and the subset of widget features available to date are those that we required. Other projects may need to implement many different capabilities depending on their needs. As well, we took the opportunity to redesign large parts of the user interface, partly to make the application look more “web-like”, but also to improve the existing interface. As such, we can’t make many useful direct comparisons between the web version and the original standalone client.

Overview of Workplace Port

The new web-based user interface for TeamWave Workplace is shown in Figure 4. Much of the user interface “around” the whiteboard has been changed from the original client, though the bulk of the program running “inside” the whiteboard remains similar.

First, some general comments are in order. The most important thing is that we were able to successfully use

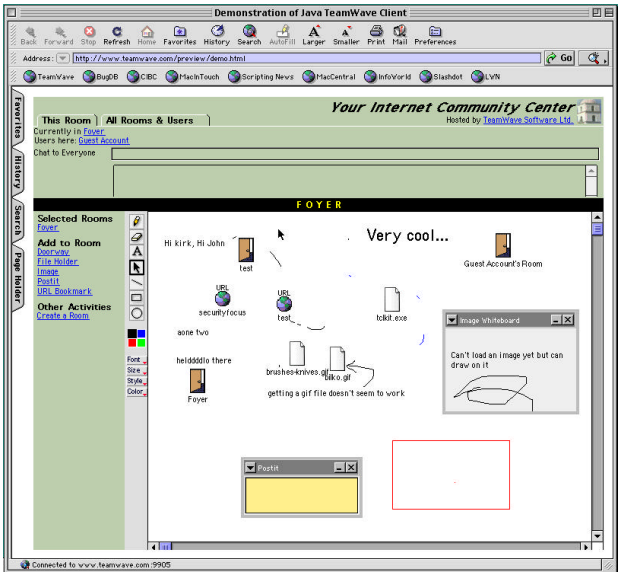

Figure 4. User interface of Web-based TeamWave Workplace.

Proxy Tk to deliver a web-based version of our application. We were able to do this very quickly, and were able to reuse the vast majority of our existing code. While many issues arose, we have been able to successfully deal with them. The applet that is downloaded to the browser is a tiny 50k in size, making it no larger than many images on web sites. From our point of view, we are very satisfied with this approach, which lets us continue to enjoy Tcl development practices while delivering web-based software.

We had to modify approximately 5-10% of the existing Tcl/Tk code to make it work with Proxy Tk. The changes were needed to address different naming for widgets, and to work around features that had not — or could not —be implemented using Proxy Tk.

The first problem was quite specific to our application. We had used multiple Tcl interpreters even within each client (e.g. for each of the tools in the whiteboard), and a fairly complex scheme whereby Tk was shared among them. When we added multiple interpreter support to Proxy Tk, we chose a simpler approach to sharing widgets between interpreters, which resulted in the names of the widgets being different. We needed to restructure the code to figure out the topmost widget name once (rather than just assuming it was “.”), and base other widget names from that. This is a convention we should have had to begin with.

In what follows, we’ll describe some of the problems and solutions required by the thin client approach used in Proxy Tk.

Network Latency

In the drawing program example, new lines are seen on the whiteboard only after a round trip of messages from the thin client to the server (to inform the server of a mouse move) and back (to draw the line). On any kind of local area network, such as a corporate intranet, this presents no problems whatsoever. On high latency networks, this can lead to very long lags in feedback to user actions, though in practice even on long-haul Internet links we have not found very severe problems. Most operations, such as selecting tools in our tool palette do not suffer from slower feedback.

This effect is minimized because most widget updates (e.g. when text is entered into an entry widget) are handled purely on the client side by the native Java widgets. In these cases, there is no interaction with the server, so no delay exists. One clear advantage of this approach over protocols like X11 or VNC is that screen refresh and most event handling is done locally on the thin client, and not sent over the network. Perceived performance is therefore comparable to running a local application.

For situations like drawing in the canvas, we have designed but not yet implemented a solution. The server could provide the thin client with a “response template” for a widget. This would be similar to a Tk binding (including percent substitutions). However, rather than it being a Tcl command, it would instead be a network message, such as the thin client might actually receive from the server. On a mouse drag in the affected widget, the thin client would use the template to create the message it would normally receive, and insert it into its own network queue. In this way, feedback is received without the need for a network round trip.

One could get arbitrarily complex with substitutions for response templates (in fact designing a complete scripting language for them!), but analyzing a number of situations where quicker performance may be warranted showed the simple case would handle most of them quite well. Essentially this technique allows us to design in application-specific performance enhancements when required.

Current Widget Contents

Unlike with standard Tk, using proxy widgets means that the current state of the widget, as visible by the user, and the state as visible by the application running on the server, may be quite different. Network messages must be sent between them to synchronize them. This has a number of implications.

First, when widgets are changed by the user, the changes must be sent back to the server. For many

widgets, e.g. checkbuttons, this can easily be done every time the widget is changed. For others, e.g. entries or text widgets, a user typing may generate a lot of network traffic; much of this may not be needed, as the application may only care about the contents of the widget after all data is entered, e.g. in a dialog.

We have added a new boolean configuration option, –immediatefeedback to entry and text widgets for this purpose. When true, changes are sent back on every keystroke. When false, changes are not sent back on every change, but only when directed to do so by another widget. As an example, buttons are set up so that before informing the server that they have been pressed, they first instruct all of their sibling widgets (i.e. those having the same parent window) to send their current value back to the server. This has proved for example an effective solution for dialog boxes, where hitting the "Okay" button ensures all widgets in the dialog box first report their current state.

Physical Size of Objects

A related issue is that the server generally does not know the physical size of widgets or objects in the thin client, particularly text objects (which rely on font metrics unknown to the server application). We'll take two examples where this became an issue.

In the first example, we wanted to display selection handles around text objects that have been selected on the whiteboard. In Tk, we do this by using the canvas' "bbox" command to determine the current bounding box of the item. With Proxy Tk, we've made the thin client send back the current bounding box of any changed text items as soon as it knows them (i.e. after a redraw). As it turns out, we never need to change the contents of the text, and immediately select the item, so we can get away with this delay. Otherwise, we'd have to wait for the bounding box to be returned before we could select the text. Note that our caching mechanism has to be clever as well; when we move the object, we can immediately update the cached bounding box, because we know just moving the object will not change its size.

A more problematic situation came up in implementing the URL-like strings shown in the user interface along the top and left. In general, these consist of one or more regions of black and blue text abutted together, with the blue parts being clickable, having status line feedback, etc.

On a regular Tk canvas, we would have created a text item for each segment, determining its starting point by looking at the bounding box of the previous segment. In Proxy Tk, we can't do this, unless we're willing to wait the round trip message after each segment.

Instead, because these URL-like strings are pervasive in the new interface, we decided to implement the behavior directly in the Proxy Tk core. Each string is created as a single text item, and we've added a new configuration option called –highlightranges. The value of this option is a list specifying what ranges of the string are highlighted, what command to invoke when they are clicked on, and a status line message for each. All processing of this option's associated behavior is handled locally by the thin client.

Client-Server Architectures

Our description of Proxy Tk has assumed that the program we're web-enabling consists of a single application that runs on the user's workstation. Workplace actually uses a client-server architecture, where the client application on the user's workstation communicates with a server program.

We could have run the Proxy Tk-based version on the server machine as a separate process, and have it make a socket connection to the actual server, running on the same machine. Instead, we took the small bit of code that accepts connections from the Java proxy and spawns new interpreters, and placed that in our existing server. So our server started up new copies of the client application in its own process, as usual with a new Tcl interpreter.

Instead of using actual socket connections (which would have worked), we developed a new Tcl channel type, loosely based on Andreas Kupries' memory channel. Our version was used to connect two Tcl interpreters in the same process together. Data written by one interpreter could be read by the other.

Extensibility and Portability

One problem with this approach, and Java in general, is that if you really need some functionality that is not available directly through Java applets, you're pretty much out of luck — no dropping down to native code.

On the other hand, running in a web browser does give us access to all kinds of facilities. Displaying a web page to the user for example is implemented as a five line Proxy Tk message handler.

To allow files to be downloaded to the user's hard drive (which can't be done directly from Java), we added a very simple httpd server to our server application. We instruct the thin client to connect to this server to download files. This server looks for URL's with a particular format (e.g. starting with /files/) and copies the file from our own file store. Images are also handled in this manner; any URL's starting with /image/ are mapped to an internal table.

Java provides the built-in routines to create an image from a URL.

Uploading files to the server (again not allowed in Java) proved trickier. The thin client is instructed to connect to our built-in httpd server, at a URL that generates an HTML form containing a file upload field. The user can then use this mechanism to choose a file from their local machine, and submit the form, which uploads the file to our httpd server. The file is then finally moved into our server file store for use by other parts of the program.

Finally, a word about portability is in order. Its no secret that despite all the hype, because of all the bugs in the various Java Virtual Machines, Java remains a "write once, debug everywhere" language. At present, our applet runs on only a small number of JVMs, though we have not yet expended the effort to make it more portable. When we do so, we expect that making our 50k applet work on different JVMs will be an order of magnitude easier than doing so for a thick client applet. To work around a bug in Java's Button widget for example, would require changing one piece of our code, not one piece for every button created by the application.

Conclusions

We've described Proxy Tk, which allows Tcl programmers to deliver web-based applications using a thin client architecture. Using an API that offers a subset of Tk functionality, a high-level Tcl program can control a 50k Java applet to implement its user interface. This technique extends the range of solutions available for delivering web-based applications using Tcl, and is particularly suited for highly interactive applications that must run in situations where downloading new software would be prohibitive.

Our experience in using Proxy Tk to develop a web-based interface for TeamWave Workplace suggests that existing Tcl/Tk code can be ported with fairly minimal changes, and that problems encountered because of the architecture are surmountable. We believe that using the techniques described here would be an effective means for developing web-based versions of a wide range of Tcl applications.

Acknowledgements

Thanks to Leo Pelland, who wrote the Java side of the prototype that led to this design and implementation. Feedback on earlier drafts of this paper was provided by Leo Pelland, Ted O'Grady, and Larry Virden.

Note to Reviewers

As of this writing, we've got a basically feature complete version running internally, with a few people besides ourselves having tried it out over long-haul networks. By the time the final version of this paper would be required for the proceedings, and certainly by the conference itself, we'll have a lot more to include and talk about regarding portability, optimization, and usage experiences.

References

1. Lam, I. and Smith, B. *Jacl: A Tcl Implementation in Java.* Proceedings of the Fifth Annual Tcl/Tk Workshop. July, 1997.
2. Levy, J. *A Tk Netscape Plugin.* Proceedings of the Fourth Annual Tcl/Tk Workshop. July, 1996.
3. Libes, D. *Writing CGI Scripts in Tcl.* Proceedings of the Fourth Annual Tcl/Tk Workshop. July, 1996.
4. Nielsen, J. *The increasing conservatism of web users.* Alertbox. March 22, 1998. http://www.useit.com/alertbox/980322.html
5. Richardson, T., Stafford-Fraser Q., Wood, K. and Hopper, A. *Virtual Network Computing.* IEEE Internet Computing 2(1), 1998. Also see http://www.uk.research.att.com/vnc/
6. Roseman, M. *Managing Complexity in TeamRooms, a Tcl-Based Internet Groupware Application.* Proceedings of the Fourth Annual Tcl/Tk Workshop. July, 1996.
7. Welch, B. and Uhler, S. *Web Enabling Applications.* Proceedings of the Fifth Annual Tcl/Tk Workshop. July, 1997.