A History of 'Tcl in the Browser'

(and a new, perhaps even better, approach)

Steve Landers
steve@digitalsmarties.com

Abstract

Tcl has been running in browsers since the early days of the Internet. And yet this deployment model is still not mainstream in the Tcl world. With the dominance of the iPad in tablet computing, and the well known limitations on scripting languages in the iOS computing environment, the need for a browser-based Tcl solution is becoming greater.

This talk will survey the various approaches to implementing Tcl in a browser, including historical solutions such as WebRouser and the venerable Tcl Plugin, Java-based solutions such as Æjaks, Javascript solutions such as IncrTcl in Javascript, and native code solutions such as NaTcl. The pros and cons of each approach will be compared, along with other approaches such as implementing the TEBC in Javascript. Finally, the talk will introduce a new effort involving the use of LLVM and the Emscripten technology to translate a Tcl interpreter (in this case, Jim Tcl) to optimized Javascript.

The Motivation

The need to run Tcl in a browser has been apparent since the early days of the Internet. But the motivation for doing so has changed over time.

In 1993 there was no scripting language for the available browsers, and it was soon recognized that Tcl could fill the void.

A team led by Mike Doyle at the Center for Knowledge Management at the University of California, San Francisco, began discussing the first web application architecture in 1993. One of the team, David Martin, knew John Ousterhout from his student days at University of California, Berkeley. Martin suggested they look at creating a Tcl interpreter (with socket communications and security added) as one of the first plug-ins, so as to provide an easy means for creating interactive content. That ultimately turned into WebWish, which was developed to run on Eolas' WebRouser in 1995 [1].

So the initial motivation was to get a scripting language in the browser, and arguably Tcl was the first. With a bit of luck and a lot less politics Tcl could have been ubiquitous. As it was, Javascript became the default scripting language when the first version of Javascript (then called LiveScript) shipped in the beta releases of Netscape Navigator in September 1995.

While Javascript is a capable language, Tcl had the added attraction of wide portability from embedded systems through to mainframe computers. It wasn't just the potential for portability of code that was the attraction, perhaps more importantly it was the portability of skills.

And so the emphasis changed from getting "a" scripting language to getting "our" scripting language in the browser. Doing so would have benefits beyond portability, including increased productivity and new deployment options.

> Anyway I know only one programming language worse than C and that is Javascript. [...] I was convinced that we needed to build-in a programming language, but the developers, Tim first, were very much opposed. It had to remain completely declarative. Maybe, but the net result is that the programming-vacuum filled itself with the most horrible kluge in the history of computing: Javascript.
>
> Robert Cailliau[2]

It is the experience of many in the Tcl/Tk community that the productivity available from Tcl-related technology is significantly more than that available from Javascript. In particular, from Tcl's string handling and Tk's development model that includes the command-based objects, the gridded geometry manager and asynchronous events with call-backs.

Deployment through a browser offers the hope of "zero install", or "minimal install" applications. In large system installations getting approval for inclusion of a new application in the Standard Operating Environment (SOE) can take years of effort, so this can be the difference between a product from a smaller developer being ignored or adopted.

But in recent years there is another, perhaps ultimately more significant, motivation: relevancy in a world increasingly focussed on mobile applications. Mobile computing has been the fastest growing area of IT for the last few years and is dominated by two platforms – Apple's iOS and Google's Android.

As has been widely reported, there are significant barriers (including technical, legal and merely perceived) to implementing applications in scripting languages on iOS. Put differently, Apple's clear preference is for native (i.e. compiled) applications to be implemented in Objective-C and scripted applications in Javascript. To that extent, the iOS Webkit-based browser (Mobile Safari) is optimized to support Javascript through technologies such as the Nitro engine [3].

It is the opinion of this author that a Tcl port to iOS is technically feasible (although significant ongoing effort would be required to implement and maintain bindings to the iOS APIs). And it would not break Apple licensing agreements to deploy applications as a Starpack [4] providing that Tcl's ability to run arbitrary code was disabled.

But even if this effort were practical (as opposed to merely feasible), there is still the issue that deployment of every application would need to go through the iTunes App Store. And herein is a significant problem: many Tcl/Tk applications are custom built for specific customers. This just doesn't fit with the App Store model.

So it seems that in the case of iOS, the only practical solution is to find a way to deploy Tcl/Tk applications in a browser.

The situation on Android is a less restrictive. The preferred application language is Java but C is supported, and there is the Scripting Layer for Android, which has allowed a number of languages to be ported. There was a port of Tcl to the Android [5] however there was no GUI support, no interface to native APIs and the installation was complicated (requiring a jail broken device). Unfortunately it appears the Tcl Android port no longer works with later Android releases.

To summarise, the motivation for Tcl in a browser, even from the earliest days of the Internet, were:

- portability (both of code and skills)
- productivity (and, in particular, the benefits of Tk)
- deployment

And to this we add the elephant in the room – mobile applications, in particular on the iPad.

The Timeline

This timeline will look at the more significant implementations of Tcl in the browser. Perhaps it would be more accurate to say deploying “Tcl applications through a browser”, because a number of these solutions still run Tcl on the server. This has implications for offline operation, which is increasingly important for mobile applications.

1995 – Eolas WebRouser

In 1995 Eolas released a version of WebRouser, an applet-enabled web browser based on Eolas' enhanced version of NCSA Mosaic that could run Tcl/Tk scripts using Eolas' WebWish Tcl plugin. WebRouser and WebWish were presented in the cover story in the February 1996 issue of Dr Dobb's Journal [6]. WebWish was the first Web Tcl implementation and one of the first plugins supported in a browser.

Pros

- Tcl and Tk
- security model
- web application support

Cons

- installation requires a plugin
- no longer available (Mosaic based)

1996 – The Tcl Plugin

In 1996 Jeff Hobbs produced a “proof of concept” Tcl plugin for Netscape following a visit to the Tcl group that was then at SunLabs. Jacob Levy (part of that group) produced the first Tcl/Tk plugin for Netscape and Laurent Demailly worked on the 2.0 implementation [7]. Version 3.0 [8] can still be installed in Firefox and Internet Explorer.

The Tcl Plugin made use of the Safe-Tcl [9] interpreter to provide a sandboxed security model. Safe-Tcl disables the Tcl commands that could potentially be harmful to the underlying system, but provides a mechanism by which these can be re-enabled in a controlled way by suitably authorised personnel.

On the positive side, the plugin still works and can be used to deploy applications on Firefox and Internet Explorer, although the installation isn't straightforward.

Pros

- Tcl and Tk
- Safe-Tcl security model
- still available on Firefox and IE

Cons

- installation – requires a plugin
- no WebKit (Safari, Chrome) port
- not available on mobile devices

1998 – Proxy Tk

In 1998 Mark Roseman and his team at TeamWave software implemented ProxyTk, a Java applet user interface toolkit for Tcl [10]. The small Java applet (50k bytes in size) ran in a browser to provide the user interface, and communicated with a Tk-like API running in a Tcl web server. Unlike the two previous browser plugin examples, in ProxyTk the application was split between the user interface running in the browser and the application Tcl code running on the server, with an efficient protocol connecting the two. The Tk commands on the server were “translated” into Java widgets in the client.

ProxyTk was ahead of its time but unfortunately it was swallowed in a corporate takeover and never reached its full potential.

Pros

- Tcl and Tk
- client / server
- real Tcl on the server

Cons

- subset of Tk features
- installation – requires a plugin
- requires Java to be enabled
- no longer available

2003 - TkWeb

In 2003 Roy Keene wrote TkWeb [11], an attempt at rendering Tcl/Tk scripts using HTML. And around the same time Wilfred J. Hansen published a technical note about Rendering Tcl/Tk

Windows as HTML [12]. Both of these were experiments aimed at proving the practicality of retaining the Tk API while rendering Tk widgets in a browser by generating HTML.

TkWeb took unmodified Tcl/Tk code and produced a Tcl/CGI application that could be run in a browser, without the need of a plugin. As such, it could be considered as the "logical" ancestor to more recent projects such as WubTk, discussed later.

Pros	Cons
• Tcl and Tk • Javascript, potential iOS support	• subset of features • experimental • no offline support

2006 – Æjaks

In 2006 Tom Poindexter developed Æjaks [13] – which "combines the server-side Ajax-based windowing system, Echo2, with the powerful simplicity of the Tcl language" . Æjaks is a thin layer over Echo2 [14] , a Java and Javascript-based platform for building interactive web-based applications. It translates Echo2 objects into Tcl objects, accessible behind a Tk-like object interface.

Æjaks used the Jacl interpreter [15] (an alternative implementation of Tcl 8.0 written in Java) so many recent Tcl features aren't available. The plan is to update Æjaks to JTcl [16], a modernised version of Jacl which provides a high degree of compatibility with Tcl 8.4.

A consideration with Æjaks is that, although Echo2 provides excellent cross-browser compatibility and uses modern Javascript techniques, it doesn't have the same amount of community acceptance or contributed widgets as other Javascript Web frameworks – in particulary jQuery [17] and jQueryUI [18].

Æjaks is a very capable system, and definitely one to consider for Tk-based web development. But being client/server it can't fully address the iPhone / iPad market.

Pros	Cons
• Tcl + Tk-like • Javascript, potential iOS support	• subset of features • no offline support

2007 – JsTcl

In 2007 Stéphane Arnold implemented JsTcl [19] , a Tcl implementation in Javascript. JsTcl was a transliteration of Picol [20], a Tcl interpreter in 550 lines of C code by Salvatore Sanfilippo.

While very limited, it did demonstrate that a Tcl interpreter in Javascript is practical.

Pros	Cons
• Javascript, potential iOS support	• experimental, with limited features

2010 – WubTk

In 2010 Colin McCormack developed WubTk, a Tk-like API that maps Tk commands run in a Tcl interpreter on a server to jQueryUI widgets running in a browser.

jQuery is a cross-browser JavaScript library designed to simplify the client-side scripting of HTML. jQueryUI is a library that "provides abstractions for low-level interaction and animation, advanced effects and high-level, themeable widgets" built on top of jQuery.

WubTk combined the approach of TkWeb (i.e. generating html) with the power and simplicity of jQueryUI.

WubTk is (as the name indicates) implemented on top of the Wub [21] pure-Tcl web server, although it isn't tied to Wub. It supports a gridded geometry manager and many common widgets. Each WubTk instance has a persistent state which is retained between requests from the browser.

WubTk has been used to deploy custom commercial applications, and was used to demonstrate a Tcl/Tk application deployed on an iPad and iPhone during August 2010: most likely the first time this occurred.

As demonstrated at the Tcl2010 conference [22], WubTk allows the integration Tcl, Tk, jQuery, HTML5 [23] and CSS3 [24]. This allows Tcl/Tk developers to use modern web features such as multimedia and 3D graphics without losing the productivity that Tk is well known for.

Pros	Cons
• Tcl and Tk • Javascript, iOS support	• subset of Tk • no offline use • Tcl only on the server-side

2011 – NaTcl

Developed in 2011 by Alexandre Ferrieux, NaTcl [25] is Tcl running in the Native Client (NaCl) [26] sandbox of the Google Chrome browser [27]. This allows the Tcl interpreter to run securely within the browser environment at speeds similar to the standard Tcl interpreters on the same platform, with full access to the Chrome DOM [28].

NaTcl was one of the first scripting language running on the Native Client. As noted in the original announcement, it is the first whose name fits well with the Google naming conventions (sodium chloride and sodium tetrachloride) [29].

NaTcl is a significant development in running Tcl within a browser. And although currently limited to Chrome, there is potential for ubiquity if the Native Client becomes accepted as a standard in the WebKit-based browser world (which includes Safari and Chrome). But that is definitely not guaranteed, given Apple's desire to control what goes on iOS.

Pros	Cons
• speed	• no Tk binding
• full Tcl	• Chrome only, so no iOS
• interface with the DOM	• plugin (of sorts)

2011 – incrTcl in Javascript

Also developed during 2011 has been incrTcl in Javascript [30], by Arnulf Wiedemann. Starting life as an enhanced version of JsTcl, it is an attempt to have a more complete version of a Tcl interpreter written in Javascript.

As well as supporting more basic Tcl features, it supports incr Tcl features and more recent features such as namespaces, dicts and the expand operator. In addition, a number of Tk widgets are supported, including implementations of TkTable, BWidget tree and the paned window.

This is an impressive effort, starting with the original 1000 lines of code in JsTcl it is currently (as of October 2011) around 22,000 lines of code.

Pros	Cons
• Tcl and Tk	• partial implementation
• Javascript, potential iOS support	• speed

2011 – NaTk

Not only does Tk provide a compelling abstraction for specifying user interfaces in native applications, as WubTk demonstrated the Tk model is also beneficial for web applications. But WubTk was tied to the client/server Wub-based architecture, so in 2011 Colin McCormack re-implemented the WubTk concepts and produced a “Tk over HTML/CSS” called NaTk, the goal being to demonstrate it with NaTcl.

While still a “proof of concept”, NaTk shows a way forward. In theory, it could be used to provide a Tk implementation for any browser-based Tcl implementation such as NaTcl or incrTcl in Javascript.

And, as has been noted previously, it goes beyond Tk allowing the development of hybrid applications that allow more advanced HTML5 features such as multimedia, CSS transformations and 3D to be used from Tcl via a Tk-like API.

Summary

So in 2011 we are left with several options for deploying Tcl applications via a browser:

- the Venerable Plug-in - not easy to deploy and is restricted to Netscape/Mozilla/Firefox or Internet Explorer (so no iOS deployment)
- Ækjaks - requires a server so no offline use and no client-side Tcl
- WubTk, also requires a server but with the advantage of HTML5 integration and iOS deployment
- NaTcl - restricted to Chrome (so no iOS deployment) and no Tk (yet)
- incrTcl in Javascript, which is still under development

Arguably none are ready for prime time, albeit they are close.

Oh no, not again[1]

With this in mind, in mid 2011 Gerald Lester, Steve Huntley and Steve Landers began discussing ways to provide Tcl/Tk on the iPad. All agreed that the only practical way is to use Javascript and map Tk onto HTML5, since a port of Tcl (let alone Tk) isn't likely to be practical.

Three basic approaches to providing Tcl were identified:

- translate application Tcl code to Javascript
- implement the TEBC engine in Javascript
- implement Tcl in Javascript

Lester was to address the first, Landers the second and all three considered the last, especially in the light of incrTcl in Javascript. This paper addresses the second approach and an unexpected development that introduced a new option for the third.

TEBC in Javascript

TEBC is an abbreviation for TclExecuteByteCode – the part of the Tcl interpreter that actually executes the Tcl Bytecodes.

1 "Curiously enough, the only thing that went through the mind of the bowl of petunias as it fell was 'Oh no, not again'. Many people have speculated that if we knew exactly why the bowl of petunias had thought that we would know a lot more about the nature of the universe than we do now" - Douglas Adams, The Hitch Hiker's Guide To The Galaxy

This approach is initially quite attractive, because it would (in theory) allow any arbitrary Tcl byte-code to be compiled using a real Tcl interpreter and then run in a browser.

After discussions with Miguel Sofer, it became apparent that there are real difficulties with this option, and it isn't worth pursuing. Perhaps that is understating the forcefulness of Miguel's argument.

The first problem is the complexity of the TEBC code. To quote Donal Fellows, '… an example of how to not write your code — TEBC currently looks like a bomb exploded in it .. but hard to do in any other way; optimized bytecode executors usually are huge balls of spaghetti' [31].

Leaving aside this complexity, there is another practical consideration: not all Tcl commands are byte-coded, and so Javascript implementations would still be needed for those. This in itself pushes the TEBC solution closer to the “Tcl in Javascript” approaches.

And finally, there is the intangible but nevertheless real concern about the constraints upon developing (or even replacing) the TEBC engine that would result from having a second widely deployed implementation “in the wild”.

The Unexpected Development

In May 2011 Fabrice Bellard announced a PC emulator written in Javascript that was fast enough to run Linux in a browser [32] "I did it for fun, just because newer JavaScript engines are fast enough to do complicated things” Bellard said [33].

The emulator is available at on Ballard's home page at http://bellard.org. Once Linux is downloaded it boots in a surprisingly fast 5 seconds when run on Safari 5.1 on a 2.8 GHz iMac i7. The Javascript code for the emulator weighs in at around 120KB. An outstanding achievement by any standard.

The technique used by Bellard was hand-coded Javascript using W3C Typed Arrays. A discussion on Stack Overflow goes into more details about the technical aspects [34].

While this development isn't directly related to the topic of Tcl in a browser, it did show that a modern Javascript interpreter is fast enough to emulate PC hardware. And if that is the case, then surely it would be fast enough to run a Tcl interpreter?

But implementing a Tcl interpreter in Javascript from scratch would be a significant project.

Fortunately, there is an alternative.

Emscripten

Steve Huntley had already flagged Emscripten [35] (a C to Javascript translator) and Steve Landers began looking into it as a way to take an existing Tcl interpreters written in C and translate it to Javascript.

The goal of the Emscripten project is simple: to allow any C/C++ code to run on the web.

Emscripten is a compiler that converts LLVM [36] bitcodes into Javascript. It can use either the llvm-gcc or clang compilers, and so supports both C and C++ (or any other language supported by these compilers).

The performance of the generated code is acceptable, without being stellar. Currently it is around 10 times slower than gcc -O3. But this will almost certainly get better with improvements over time to LLVM, Emscripten, Javascript optimisers and Javascript engines.

Already a number of languages have been ported to Javascript using Emscripten, including Python, Ruby and Lua. The goal became to add Tcl to that list.

The first decision was which Tcl to use. While the core Tcl release is an obvious choice, there are a number of potential problems in using it for this project:

- it is a relatively large code base
- it contains feature that are not relevant to a browser environment (e.g. threads, file I/O)
- it isn't easily modularised

One of the small Tcl interpreters like Picol would have also been an option, but this would have been too limiting.[2]

But there is an alternative that is modular, small and with most key Tcl features: Jim Tcl.

Jim Tcl

Originally developed by Salvatore Sanfilippo, and now maintained by Steve Bennett, Jim is a small footprint re-implementation of Tcl that is particularly suited for embedded environments [37].

Jim implements a large subset of Tcl and adds many advanced features like references with garbage collection, closures, a built-in Object Oriented programming system, functional programming commands, and first class arrays.

Jim is also quite small, around 10k lines of code and a binary size of between 100-200 kB depending on the modules used.

Jim passes many Tcl unit tests, and many Tcl programs run unmodified, but it is best to think of Jim's relationship with Tcl as one of programmer portability than necessarily program portability.

So the plan became to treat the browser as an embedded system and to compile Jim Tcl to Javascript using Emscripten.

---

2 the author has subsequently found that Tom Poindexter built Picol using Emscripten at about the same time

Jim JS

Clearly the project needs a better name than Jim JS but (in a move somewhat atypical for an open source project) the decision was made to actually get the software working instead of investing time and effort in a cute name or flashy website. But I digress …

Emscripten requires specific versions of LLVM and Clang or LLVM-GCC, along with one or both of the SpiderMonkey or V8 Javascript engines. Given these specific requirements an Ubuntu virtual machine was created to avoid clashes with existing compiler toolchains, and also to facilitate sharing the development environment. As an aside, the ability to check-point and restart the virtual machine (a feature of VMware and other products) greatly facilitated the testing of the configuration.

Once installed, it was a matter of selecting the Jim Tcl modules, creating a build script that calls the Jim Tcl Makefile as appropriate, then waiting for it to break.

And break it did. But fortunately there are good examples in the Python build system, plus Emscripten includes a make proxy tool that converts normal build commands to those appropriate for Emscripten. The result was 2.4 MB of rather dense and obtuse Javascript.

Invoking the generated Javascript in a browser is relatively straight forward: define an html form containing an input field for the Tcl code, along with two Javascript functions: one to evaluate the entered text by passing it to a predefined function in the generated Javascript, one to print the results.

```
function execute(text) {
  printed = false;
  Module.run(text);
  if (!printed) {
    print('<small><i>(no output)</i></small>');
  }
}

function print(text) {
  console.log(text);
  var output = document.getElementById('output');
  if (output) output.innerHTML += text + '<br>';
  printed = true;
}
```

In a typical application (rather than a test environment) the Module.run() function would be invoked from Javascript, either directly or as the result of an AJAX operation. The generated Javascript can make calls to any Javascript function, and so it would be straightforward to add a facility to make DOM calls from Tcl.

In early tests it was found that simple commands like “set i 10” worked, but compound commands like “set i 10 ; set j 20” did not, nor did expr and many other commands. The root cause seemed to be memory management issues, as a diagnostic from the Emscripten runtime support code flagged that something was trying to allocate zero btyes.

A discussion with Steve Bennett quickly identified the problem: rather than test that the requested allocation size is not zero, Jim Tcl relies on the memory management library to return zero bytes if it is. Once this was identified it was easily fixed in the Jim JS runtime code.

The next issue was that many commands worked, but others such as expr caused Jim to exit with no error message. The solution was found by enabling various DEBUG_SHOW options within the Jim Tcl C headers. This caused Jim to display debugging information:

For example:

```
command = expr 1
==== Tokens ====
[ 0]@1 ESC 'expr'
[ 1]@1 SEP ' '
[ 2]@1 ESC '1'
[ 3]@1 EOF ''
==== Script ====
[ 0] LIN
[ 1] ESC expr
[ 2] ESC 1
==== Expr Tokens ====
[ 0]@0 INT '1'
[ 1]@0 EOL ''
_strtoull is not a function              jim.js:32571
```

This shows that a C library function (strtoull) has not been implemented in the Emscripten Javascript runtime, but by undefining HAVE_LONG_LONG when building Jim the equivalent (and implemented) strtoul is generated.

Futher testing showed there were several runtime functions that were partially implemented. For example, not all strtol(3) parameters are supported. The solution is to implement a wrapper function in Javascript that accepts the calls used by Jim Tcl and maps them to the appropriate Emscripten runtime function.

This shows the basic steps in finishing the port of Jim Tcl to Javascript:

- run a test and look for the missing Javascript functions
- adjust the Jim configuration to avoid generating missing functions
    - or implement a wrapper function in Javascript to convert to existing but incompatible runtime functions
    - or implement a new function in Javascript

At this point in time (October 2011) this is an ongoing activity.

Performance and Optimization

Anecdotally, performance is quite acceptable. For example, testing on an iMac 2.8 GHz i7 results in the following using ActiveTcl 8.6b1.2 and Jim in the browser:

time {set a 10} 100000

ActiveTcl 8.6b1.2	0.25151566 microseconds per iteration
Jim/Firefox	18 microseconds per iteration
Jim/Safari	16 microseconds per iteration

time {set a 10; set b $a} 100000

ActiveTcl 8.6b1.2	0.42748254 microseconds per iteration
Jim/Firefox	30 microseconds per iteration
Jim/Safari	27 microseconds per iteration

So that's 60-80 times slower than "native" (let's say an order of magnitude) on a simple operation, with essentially no optimization.

As mentioned, the generated Jim Javascript code is around 2.4 Mb in size. Whilst this isn't outrageous, it can be improved significantly by running it through a Javascript optimizer such as Google's Closure compiler []. The resulting jim.js is around 650 kB, it obviously loaded more quickly but was not measurably faster to execute. Other Javascript optimizers are yet to be tried.

Summary and Conclusions

The need for Tcl/Tk deployment in a browser is no less now than it was in the early nineties.

And in spite of several efforts over the years the average Tcl developer is still not in a position to deploy an application in a browser. But with the growth in mobile computing, and in particular iOS, the need has never been greater.

There are at least three projects underway that could meet this need in varying degrees (NaTcl, Jim JS and incrTcl in Javascript). And in many ways all three are complementary.

If one was to crystal-ball gaze, there is a scenario where Jim JS is used to get ubiquity, NaTcl for the case when performance is needed, with parts of incrTcl in Javascript for it's excellent widget support.

But dreaming even more, imagine a hand-crafted, fast Javascript implementation of Tcl (that took the lead from the Linux in a Browser project) combined with a more complete NaTk (providing Tk over HTML and HTML5/CSS3 integration).

And not just for deploying applications in a browser. As shown by Entice [38] and other tools, desktop applications with embedded browsers are both viable and attractive. So perhaps the next generation of Tk should not be based on X11, Windows or Cocoa, but on Javascript, HTML and CSS.

Tk9 anyone?

References

[1] WebRouser Announcement – http://1997.webhistory.org/www.lists/www-talk.1995q3/0566.html
[2] Interview with Robert Cailliau - http://en.wikinews.org/wiki/Wikinews:Story_preparation/Interview_with_Robert_Cailliau
[3] Javascript Engine - http://en.wikipedia.org/wiki/JavaScript_engine
[4] Starpack – http://wiki.tcl.tk/3663
[5] Tcl Android – http://wiki.tcl.tk/27643
[6] WebRouser – Dr Dobb's Journal, Issue #244, February 1996

[7] Levy, J. A *Tk Netscape Plugin*. Proceedings of the Fourth Annual Tcl/Tk Workshop. July, 1996. http://www.usenix.org/publications/library/proceedings/tcl96/full_papers/levy/index.html
[8] Tcl/Tk Plugin Version 3 – http://www.tcl.tk/software/plugin/
[9] Safe-Tcl – http://labs.oracle.com/techrep/1997/smli_tr-97-60.pdf

[10] Roseman, Mark *Proxy Tk: A Java applet user interface toolkit for Tcl*. Proceedings of the Seventh Annual Tcl/Tk Conference, February 2000, http://www.usenix.org/events/tcl2k/full_papers/roseman/roseman_html/
[11] TkWeb – http://www.rkeene.org/projects/tkweb/
[12] Hansen, Wilfred J. *Rendering Tcl/Tk Windows as HTML*. Proceedings of the Tenth Annual Tcl/Tk Conference, July 2003 http://www.tcl.tk/community/tcl2004/Tcl2003papers/rendering.doc
[13] Æjaks – http://aejaks.sourceforge.net/Aejaks_Home
[14] Echo2 – http://echo.nextapp.com/site/echo2
[15] Jacl – http://wiki.tcl.tk/1637
[16] JTcl – http://jtcl.kenai.com/
[17] JQuery – http://jquery.com
[18] JQueryUI – http://jqueryui.com
[19] JsTcl – http://wiki.tcl.tk/17972
[20] Picol – http://wiki.tcl.tk/17893
[21] Wub – http://wiki.tcl.tk/wub
[22] Landers, Steve *WubTk - Tcl/Tk Apps Anywhere*. Proceedings of the Seventeenth Annual Tcl/Tcl Conference, October 2010, http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2010/SteveLanders/WubTk/wubtk.pdf
[23] HTML5 – http://en.wikipedia.org/wiki/HTML5
[24] CSS3 – http://en.wikipedia.org/wiki/CSS3
[25] NaTcl – http://wiki.tcl.tk/28211
[26] Google Native Client – http://en.wikipedia.org/wiki/Google_Native_Client
[27] Google Chrome – http://www.google.com/chrome
[28] DOM – http://en.wikipedia.org/wiki/Document_Object_Model
[29] Google native code browser plug-in gets tickled – The Register, April 14, 2011 http://www.theregister.co.uk/2011/04/14/tcl_on_native_client/
[30] incrTcl in Javascript – http://wiki.tcl.tk/28293
[31] TEBC – http://wiki.tcl.tk/22133
[32] Fabrice Bellard – Javascript PC Emulator – Technical Notes - http://bellard.org/jslinux/tech.html
[33] cnet News - Javascript: Now powerful enough to run Linux, May 17, 2011, http://news.cnet.com/8301-30685_3-20063563-264.html

[34] StackOverflow discussion on Bellard's performance tricks – http://stackoverflow.com/q/6245191
[35] Emscripten – http://github.com/kripken/emscripten/wiki
[36] LLVM - http://llvm.org/
[37] Jim Tcl – http://jim.berlios.de/
[38] Landers, Steve *Entice – Embedding Firefox in Tk* . Proceedings of the Thirteenth Annual Tcl/Tcl Conference, October 2006 - http://www.tcl.tk/community/tcl2007/papers/Steve_Landers/Entice.pdf