Ten Years of Rapid Development

Mark Roseman
mark@markroseman.com

Abstract

Over the past ten years, Tcl has been a key development tool for a range of collaborative systems, ranging from academic research prototypes to large-scale use in commercial web conferencing products. This paper examines how this use of Tcl evolved, highlighting some lessons for developers of growing systems, and focusing on how well Tcl can support traditional software engineering practice. Tcl was also made a focus of technical due diligence during a company acquisition, and many of the positions offered in its support may prove useful to others facing management pressure over their choice of development tool.

1. If Development was so Rapid…

For the past ten years, I've been involved in the development of a series of collaborative systems built using Tcl. These systems encompassed the spectrum from a Tcl extension to aid in academic research prototyping, to several large, robust desktop and web-based commercial products.

Like so much Tcl development work, very small and focused development teams built these various systems, even as some of the software grew larger and more complex. One constant theme was experimentation and reinvention; a wide variety of systems evolved over the ten-year period. It's safe to say that some of the abrupt corner-turns, particularly during the dot-com frenzy, necessitated a constant stream of rapid redevelopment and change.

This paper will cover several phases of the work. It began around 1992 with GroupKit, a university-developed Tcl extension used for rapid prototyping of collaborative interfaces and experimentation with underlying software architectures. A follow-on application in 1996, TeamRooms, was a more robust desktop collaborative environment combining a range of tools. TeamRooms was later spun-off into a company I founded, TeamWave Software Ltd., and the software was further developed as TeamWave Workplace. TeamWave moved to web-browser-based products in 1999, applying the technology to virtual communities, music sharing, e-learning, and web conferencing/meetings.

The company was eventually acquired in late 2000; our use of Tcl was a key issue during the technical due diligence phase leading up to the acquisition. Throughout this paper, I'll highlight many of the issues that we used to pitch Tcl as a significant strength, rather than a liability for the acquirer.

While some of the issues are familiar (e.g. rapid development time, easy integration), others are less obvious, such as arguments surrounding performance and scalability. These can depend heavily on Tcl development practices, and effectively using Tcl in support of sound software engineering practices. While best practices aren't always obvious, and there are some issues with practical use in organizations, all the benefits certainly made it an excellent choice at every phase of our development work.

1.1. Experience papers

Earlier years of this conference have brought a variety of experience papers, most notably Don Libes' paper reflecting on seven years of evolution of the Expect extension [5], and Tom Phelps' paper on building the TkMan application [9].

Through these and other papers, we've learned practical techniques for common problems, more obscure issues that creep up during longer-term deployment, and examined the effect of aesthetic language issues and design choices. Through war stories and experiences, we can avoid pitfalls and discover new practices to aid in our own development, and understand better some of the factors affecting core language evolution.

This paper differs somewhat from previous experience papers. It covers a wider range of systems, examines software engineering practices with Tcl as code and teams start to grow, and addresses some of the "soft" concerns such as organizational pressures as they impact the choice of development tools and practices.

1.2. Collaborative Systems

The application domain I've been working in is collaborative systems, a rather broad area that covers a range of things from chat systems, to email, to bulletin boards, to video conferencing and more. Other names often applied to the same domain include "groupware" and "conferencing".

These systems are shared by several people over a network, and tend to be classified according to whether they allow people to work together at the same time such as in a meeting ("real-time" or "synchronous") or if they are to support people working together over a longer period of time at different times ("asynchronous").

Most of my work has been in more visually oriented systems, often focused around visual surfaces like whiteboards or slide shows. These tend to be highly interactive areas, where gestures, annotations, object manipulation, and other small actions must be quickly communicated to everyone sharing the system.

Developing these systems is interesting, because not only do they involve many complex technical issues (network communications, data storage, security, etc.) but because people are actually using these tools to try to communicate with each other. That means that user interfaces (and even social context) become very important.

It is no surprise then that collaborative systems share many characteristics with other computational, client-server, networked or user interface applications commonly developed with Tcl/Tk.

2. GroupKit: Prototyping Interfaces

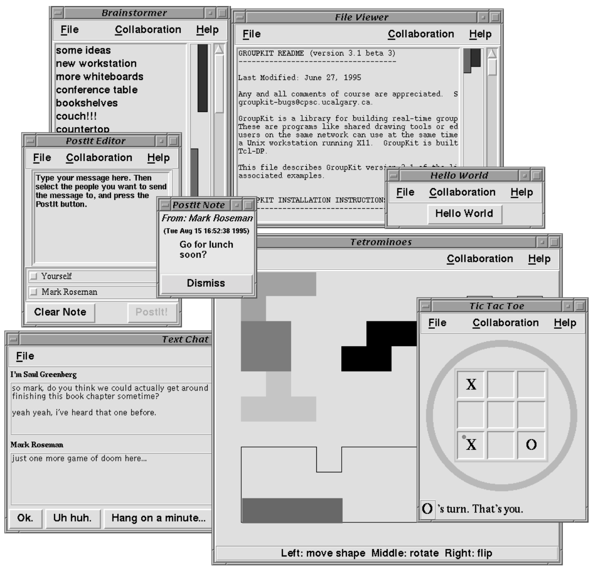
GroupKit [10,13] was an open source Tcl/Tk extension developed at the University of Calgary, providing common collaboration facilities to aid in the prototyping and evaluation of groupware applications and architectures. It was used by many students and researchers at various institutions. Figure 1 shows some examples of the types of tools prototyped with GroupKit.

GroupKit had actually started life as a C++ class library, based on the InterViews GUI library. The move to Tcl/Tk (precipitated by some eager experimentation immediately after the first release of Tcl-DP) took about one month and yielded some benefits over the C++ version by now very familiar to most Tcl/Tk users:

- The easier learning curve of Tcl/Tk made the system accessible to more people; ramp-up time for those unfamiliar with Tcl was measured in hours or days, not weeks, making GroupKit accessible to coursework for students.
- The rich Tk widget set, especially the canvas and text widgets, enabled more sophisticated interfaces to be prototyped more easily and quickly.
- Code size, development time, and debugging time shrunk by an order of magnitude for typical applications, which was critical for the target audience.
- We were later able to develop Windows and Macintosh ports of GroupKit, supplementing the original Unix version.

As a result, the system gained significantly more widespread use in the research community, more interesting systems were built, we could add many more features, make it flexible enough for a wider range of uses, and do more of our own experimentation.

In many ways, this was a good example of the traditional use of Tcl/Tk, making prototypes and small applications, developed by very small numbers of often-novice developers. Using Tcl/Tk helped bring down the scale of larger problems. It let developers think at a higher level about the problem, and reduce the amount of code needed to express the solution. As a result, problems were solvable more quickly and easily by individual developers.



some ideas
new workstation
more whiteboards
conference table
bookshelves
couch!!!
countertop

```
GROUPKIT README (version 3.1 beta 3)
------------------------------------

Last Modified: June 27, 1995

Any and all comments of course are appreciated.  S
groupkit-bugs@cpsc.ucalgary.ca.

GroupKit is a library for building real-time group
These are programs like shared drawing tools or ed
users on the same network can use at the same time
a Unix workstation running X11.  GroupKit is built
Tcl-DP.

This file describes GroupKit vers
associated examples.

PKIT INSTALLATION INSTRUCTION
```

Type your message here. Then select the people you want to send the message to, and press the PostIt button.

Yourself
Mark Roseman

From: Mark Roseman
(Tue Aug 15 16:52:38 1995)
Go for lunch soon?

Hello World

I'm Saul Greenberg
so mark, do you think we could actually get around finishing this book chapter sometime?

yeah yeah, i've heard that one before.

Mark Roseman
just one more game of doom here...

O 's turn. That's you.

Left: move shape Middle: rotate Right: flip

Figure 1. Examples of interfaces developed with GroupKit.

3. TeamRooms: A Larger Application

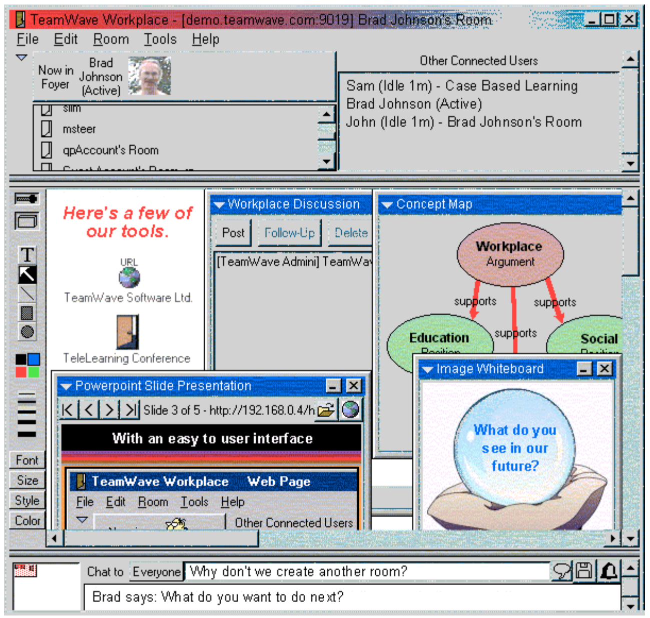
TeamRooms [11], and later the commercial TeamWave Workplace, was a full-fledged collaborative application, rather than a prototyping environment. It provided members of a group a set of electronic rooms where they could share work, both in real-time and asynchronously. The rooms combined a variety of different shared tools. Figure 2 shows an example.

These systems relied on a central server, along with a desktop application that ran on each client's machine. Both client and server ran on Mac, Windows and several Unix platforms. Internally, the systems were fairly standard Tcl/Tk applications. A novel feature was that the client used a number of different Tcl interpreters to juggle the different tools in the environment, allowing new tools to be added, yet from a programmer's point of view appear isolated from the rest.

The application, though considerably larger and more complex than earlier work in GroupKit, was still reasonably sized (<50k loc), and was built primarily by 2-3 developers. Development challenges primarily surrounded then-bleeding-edge features in Tcl, such as the cross-platform versions of Tk and the core's new I/O model and socket support. Performance requirements dictated a modest amount of spot recoding in C and other isolated optimizations, all easily identified using standard profiling.

TeamWave Workplace was again typical of many more mature Tcl/Tk applications, becoming more robust and full-featured with each version, but still developed on a fairly small scale. The relatively small code base and size of the developer team allowed ad hoc development practices to suffice. The coding style (e.g. documentation, naming conventions) was fairly loose, inter-module dependencies were high, and testing was largely manual.

This looseness was conducive to the rapid changes that occurred, especially in early versions of the commercial system, when many new features were added to meet customer needs. But as we'll see soon, the ad hoc development practices started causing problems.



Now in Foyer
Brad Johnson (Active)
slim
msteer
qpAccount's Room

Other Connected Users
Sam (Idle 1m) - Case Based Learning
Brad Johnson (Active)
John (Idle 1m) - Brad Johnson's Room

[TeamWave Admin] TeamWa

Slide 3 of 5 - http://192.168.0.4/h
With an easy to user interface

Why don't we create another room?
Brad says: What do you want to do next?

Figure 2. Example of TeamWave Workplace interface.

4. Moving to the Web

By late 1998 it became clear that the TeamWave application had to be retargeted to run inside a web browser, rather than as a double-clickable desktop application. For many of our target customers in education and virtual communities, the need to download software and run another application was too large a barrier for new users, and an impediment to regular use.

While modest angel funding (by dot-com standards!) had allowed us to grow our development team to 5-10 people, trying to do a complete rewrite (e.g. as a conventional Java applet) was both too large a project and too risky given the state of client-side Java at the time. We wanted a solution that would preserve as much of our code base as possible, and enable us to continue our rapid development using Tcl if we could.

4.1. Proxy Tk

The solution we came up with involved development of a Tk replacement that, rather than talking to the local UI toolkit, communicated with a very small Java applet running in the client's browser. The client application using Tk, formerly running as an application on the user's desktop, migrated to a new Tcl interpreter on the server machine.

The extension we developed, called Proxy Tk [12], let us continue primary development in Tcl/Tk, isolate the considerable number of Java issues in a tiny body of code, and run (from the user's perspective) download-free. Proxy Tk proved a fantastic success.

The core collaboration environment went through a number of iterations once it hit the web version, supporting a diverse range of uses from virtual community and music sharing, e-learning, web presentations, online meetings, and more (see Figure 3 for a few screenshots). Without the rapid development advantages that Tcl afforded, it is clear we never could have explored these areas with the size of team we had.

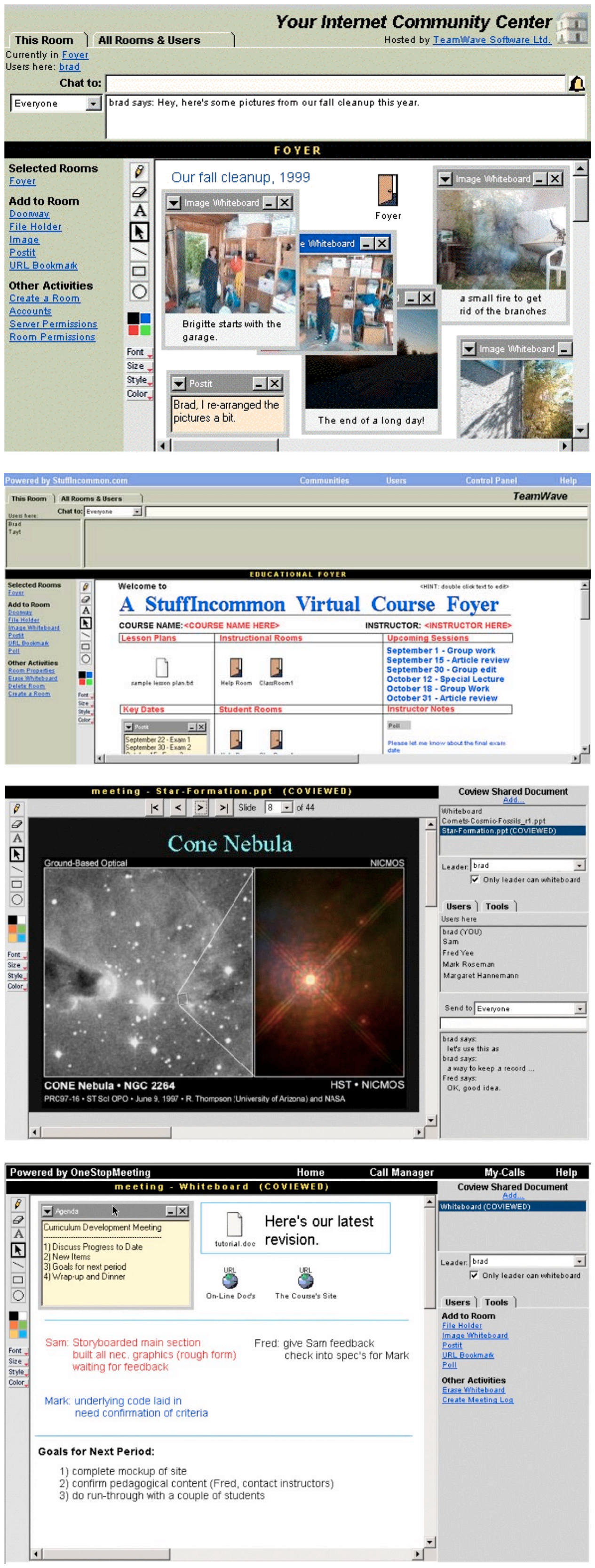
4.2. Site Management

The move to web-based systems also moved the product from a model of supporting a single workgroup on a server to an "application service provider" model, where we were hosting very large numbers of virtual meeting rooms or classrooms for a large number of different groups.

We therefore needed to develop a typical web site infrastructure, supporting registering users and associated user management tasks, creating and managing meeting rooms or courses, work groups, meeting scheduling and emailed invitations, and more.

Early versions of this portion of the system were developed using Apache and CGI, using the cgi.tcl library [4] as an aid. We later moved to a more powerful infrastructure, using AOLserver. For both variants, we kept a separate centralized Tcl process for storing system wide data using Metakit [14], as well as interacting with the virtual meeting room processes.

Again, this portion of the system went through considerable change and evolution as we explored different applications, and Tcl sped things up greatly. There weren't many technical obstacles here, as we used fairly standard web techniques.



brad says: Hey, here's some pictures from our fall cleanup this year.

Our fall cleanup, 1999

Brigitte starts with the garage.

a small fire to get rid of the branches

Brad, I re-arranged the pictures a bit.

The end of a long day!

Welcome to
A StuffIncommon Virtual Course Foyer
COURSE NAME: <COURSE NAME HERE>
INSTRUCTOR: <INSTRUCTOR HERE>
Lesson Plans | Instructional Rooms | Upcoming Sessions
September 1 - Group work
September 15 - Article review
September 30 - Group edit
October 12 - Special Lecture
October 18 - Group Work
October 31 - Article review
Key Dates | Student Rooms | Instructor Notes
September 22 - Exam 1
September 30 - Exam 2
Please let me know about the final exam date

Cone Nebula
Ground-Based Optical
NICMOS
CONE Nebula • NGC 2264
HST • NICMOS
PRC97-16 • ST ScI OPO • June 9, 1997 • R. Thompson (University of Arizona) and NASA

Whiteboard
Comets-Cosmic-Fossils_r1.ppt
Star-Formation.ppt (COVIEWED)

Users here
brad (YOU)
Sam
Fred Yee
Mark Roseman
Margaret Hannemann

brad says:
let's use this as
brad says:
a way to keep a record ...
Fred says:
OK, good idea.

Curriculum Development Meeting
1) Discuss Progress to Date
2) New Items
3) Goals for next period
4) Wrap-up and Dinner

tutorial.doc Here's our latest revision.

Sam: Storyboarded main section
built all nec. graphics (rough form)
waiting for feedback

Fred: give Sam feedback
check into spec's for Mark

Mark: underlying code laid in
need confirmation of criteria

Goals for Next Period:
1) complete mockup of site
2) confirm pedagogical content (Fred, contact instructors)
3) do run-through with a couple of students

Whiteboard (COVIEWED)
Add to Room
File Holder
Image Whiteboard
Postit
URL Bookmark
Poll
Other Activities
Erase Whiteboard
Create Meeting Log

Figure 3. Images of some of TeamWave's web-based systems.

4.3. Growing Pains

Both in the later versions of Workplace, and throughout the different versions of the web-based systems, several cracks began emerging in our ad hoc development practices. The problems became exacerbated as we brought on new developers, and as the code base grew (later versions of the web-based systems reached approximately 90k loc Tcl, 35k loc C, and 10k loc Java).

Almost without exception, our new developers had no problem picking up Tcl very quickly. One new hire, a previous junior C/C++ programmer, tasked with learning Tcl and developing an automated build system for our software we figured would take two weeks, returned in two days with the completed version. Another team member, without a formal computer science background, was happily hacking out web scripts in no time.

Getting new code working reliably with our existing code proved somewhat more difficult. Poor module boundaries made it hard to find the needed calls to make, a problem exacerbated by spotty documentation. A lack of established coding conventions soon led to a plethora of variable and procedure naming styles, module API's, and more.

Not surprisingly, adding new features tended to break some of the old code, leading to greatly increased manual testing time. Increasingly, work fell into the pattern of "two steps forward, one step back." The pace of development slowed considerably.

This hardly came as a shock of course; we didn't expect all of Tcl's rapid development features to give us maintenance-free code, or to automatically solve all the problems of managing growing software systems.

These problems did take longer to arise (since the code base in Tcl was smaller than it would be in conventional languages), and as we'll see in the next section, Tcl itself made it easy to address these problems.

5. Software Engineering with Tcl

As we've seen, Tcl is not immune to the problems faced in every other language as programs and teams grow in size. Luckily, standard "motherhood" approaches to these problems (e.g. [6,7]), as well as more recent agile development approaches [1] also happen to work just fine in Tcl. In some cases, Tcl may in fact be particularly advantageous for implementing such solutions.

Tcl doesn't force you to write clean, easy-to-maintain software, but it doesn't prevent it. This section will highlight a few of the techniques we applied as we started encountering some of the problems discussed earlier. Obviously, applying these techniques in advance of encountering problems or in their early stages is greatly preferable to trying to apply them while in the throws of a full-blown crisis!

5.1. Coding style

There are lots of good reasons for having a uniform low-level coding style, including such things as documentation standards, naming schemes for variables, procedures, modules, etc., spacing, tabs and bracketing standards, and all the other minutiae of syntax. Developers can think about the problem at hand rather than formatting, find what they're looking for easier, and make all code truly a group resource, avoiding the "my code, your code" problems.

Luckily, Tcl itself offers a fantastic example of a code base that is superbly engineered and follows exacting coding conventions. There are excellent guides for writing both C code [8] and Tcl script libraries [3]. Use them directly or adapt them to your own needs.

Adopting and enforcing a uniform coding style was a no-brainer, which two long weeks of RSI-inducing monotony suggested should have been done much earlier.

5.2. Modularity

Good module design is another important factor as programs grow larger. While in small prototypes it is reasonable and even advantageous to call routines from all over the place, manipulate global data and so on, the poor coupling and cohesion inherent in this approach clearly doesn't scale.

Namespaces and packages, as well as object systems such as [incr Tcl], offer numerous aids to help build Tcl modules. We chose to implement a scheme patterned after Tcl core commands such as 'string' or 'file'. Each module, in its own file, implements a single top-level Tcl command. The first argument to the command is always a subcommand specifying the actual operation to invoke, followed by needed arguments. While the top-level command will invoke other procedures within the module to complete the command, all callers outside the module only go through the single public interface. All module data is stored in an array having the same name as the module.

This gives us the expected benefits: a clear, well-defined interface, API documentation easily visible at the top of the file, a single entry point to the entire module, easy inspection of data, no naming collisions, and more.

Structuring modules in this way also made it easy to move an entire code module into C, when required for performance or other reasons, without impacting any other code in the system. Provided the C version of the command implemented the same API, everything continued to work. This removed many obstacles as the system evolved.

5.3. Automated testing

It is hardly a secret that Tcl is a popular environment for writing automated tests. Large companies like Motorola, Cisco, Oracle and Sybase (e.g. [2]) have millions of lines of Tcl test scripts. Tcl is also the basis for testing frameworks like DejaGnu, and the "tcltest" framework bundled with Tcl itself, which we used.

As agile development enthusiasts will attest, automated tests can speed up code development, and certainly catch regression errors that are introduced as changes are made to existing code. We didn't make as much use of automated tests as we should have, except for key modules where data integrity was critical. This cost us dearly in terms of re-introduced bugs and exceedingly laborious manual test plans that had to be developed and frequently run through.

Tcl makes testing easier than in other languages, and well-defined modules are very amenable to automated unit tests. We also used it to write broader functional tests that interacted with web pages in our applications. Having the test cases and code being written in the same language is also a tremendous aid to developers, who have one less thing to learn, and therefore one less excuse not to write test cases.

5.4. Build scripts

Automated build scripts, which completely check out (you do use version control, right?) and build your software from scratch (and run through all your automated test cases!), are also a standard technique for reducing errors. Particularly when automatically run every night, they quickly weed out errors introduced in the code, and ensure a reliable build.

Again, because Tcl is exceptionally good for writing scripts that control other programs, it is ideal for build scripts. As with test cases, because these scripts are written using the same tool as the main code, any member of the team can extend, debug or fix the build scripts as needed.

5.5. Best practices

As Tcl applications grow, it is gratifying to know that traditional software engineering techniques will help manage that growth, and perhaps even be easier to apply than in many conventional languages. Later sections will also show some other benefits of these techniques.

It is clear though that such management does require a dedicated effort, and that while Tcl will make such a task easier, it does not come free. Applying techniques like those described here, along with code and architecture reviews, preparing design documents, test plans, and other best practices, are critical as programs and teams grow.

An obvious corollary to this extra process is that the 5-10 times developer efficiency increase afforded by a scripting language like Tcl over conventional languages cannot be sustained as program and team size increases. The smaller code size of Tcl programs does however mean that it will take a lot longer before programs become "large", and developer productivity begins hitting that wall. Of course, extra process must be added for larger programs developed in other languages too, slowing them down as well.

Though we did not measure it, I'd estimate overall efficiency was still at least 3-4 times higher with Tcl, across the entire team. Of course, without the extra process, efficiency would have quickly approached zero. Bugs would have continued to be introduced, senior developers would have their entire time occupied with junior developers lost in the code, and so on.

6. Tcl in the Hot Seat

In mid-2000, we began searching for a company to acquire TeamWave. We had discussions with a number of companies, and eventually were acquired in late 2000 by a Boston company now called Sonexis, a developer of audio conferencing technologies complementary to our own data conferencing software. At that point, they were a fairly typical smaller (~200 people) software company, with a very Windows/C++/COM/ASP based development culture.

Throughout technical due diligence, both leading up to the acquisition by Sonexis, and in our discussions with other companies, our use of Tcl as a primary development language was certainly one of the big questions everyone had. It fell to us to communicate why Tcl was an asset to us, and not a liability. We also needed to explain how our systems would be able to easily and effectively integrate into other existing systems based on very different technologies.

The question of using Tcl was only one of many questions of course; the underlying issues are always whether the team and technology would be both a good fit and enhance existing assets. Taken in that light, this wasn't so much a "defense" of Tcl, as communicating the positive aspects of our team, tools, technologies and development practices. The message was that Tcl was chosen to be part of our entire development effort because of the concrete advantages it offered.

I'll describe both some of the obvious strengths of Tcl and our refutations to some common technical and non-technical misconceptions that arise, and then go into considerable detail regarding four of the key technical advantages we offered: integration, configurability, reliability, and performance and scalability.

While some aspects of these discussions are particular to our technology, many of these issues are both common and generally applicable to many systems.

6.1. Obvious Tcl Strengths

There were a few clear advantages developing our systems in Tcl gave us that are quite familiar to most people, and require little discussion here.

Higher level programming. As a scripting language, Tcl programs require less code, less housekeeping, and don't have the same edit-compile-debug cycle as conventional languages. It is therefore possible to attack development projects using Tcl with less people or time. This was easily demonstrated by looking at what our application did, compared with the volume of code and the amount of time and number of people we'd needed to develop it. Similarly, walking through some of the bits such as user interface and networking that are more complex in other environments was a convincing demonstration.

Cross platform. For various organizations, the ability to move between different Unix variants, or to move from Unix to Windows was important. Versions of Workplace had been fully cross-platform, and we'd done experimental ports of the web-based products to Windows. Again, demonstrating what we had and explaining the small amount of changes required was important here.

Easy licensing. Some people we had discussions with were happy that Tcl's open source licensing made distribution both cost free and hassle free. In most cases, this becomes just one less thing to have to deal with.

6.2. Common Misconceptions

Some common myths about Tcl were also fairly easy to deal with. With all of these, actual demonstration rather than theoretical arguments was key.

Tcl is unstructured. As with any language, Tcl certainly can be unstructured. A brief skim through our code, rigorously structured and coded according to strict style guidelines, was more than sufficient to prove it doesn't have to be. The related issue is that Tcl is only good for small programs; again, we were able to demonstrate with our system that that wasn't the case.

Tcl is hard to hire for. It's not that common to find Tcl on a resume. Our argument was simply that most competent developers could pick Tcl up very quickly. We explained the backgrounds and learning curves for our own developers, and did a short walkthrough of both some simple and more complex code, highlighting the similarities with other languages.

Tcl is unsupported. This came up infrequently, and was more related to issues of how mainstream was it rather than actual support. On that front, we could quickly point to large companies like AOL, NBC, Cisco, Motorola and many others. On the actual support front, the open code base, availability of consultants and evidence of the active Tcl developer community, coupled with poor support experiences with many proprietary tools, were convincing enough.

7. Advantages for Larger Systems

This section describes four key areas that received significant attention during our various discussions: integration, configurability, reliability, and performance and scalability. A common thread is that these are all significant issues in developing larger systems.

7.1. Integration

Integration is a critical issue when incorporating any new technology into an existing product. But integration is also an issue in terms of accessing code libraries or other facilities not built into the language. For both these cases, it is not surprising that Tcl's recognized strength as a "glue" language was a strong asset.

Platform support. Our web applications were hosted on Unix; the ability to easily port to Windows (or other Unix variants) removed an obvious obstacle to integrating with existing code that was running on Windows.

Library integration. Obviously, not every possible feature is built into any programming language; Tcl in particular has tended to keep many features out of the core language, leaving them to be provided as extensions. As well, most third party libraries will have bindings for C/C++ and not Tcl. This inevitably leads to "but Tcl doesn't have feature X" arguments.

The ability to call C/C++ code from Tcl is the obvious response to these issues. We already relied on several extensions, as well as a body of our own C code accessed with Tcl commands. Tcl handles this noticeably better than most other languages, and this ease is readily demonstrated.

COM. Several of the companies we talked with had a large body of code that used COM for internal communications, and it was expected that we would want to integrate at that level. Despite Tcl not having built in COM support, we were able to demonstrate how it could work with extensions (e.g. TCOM). We already used TCOM in one small piece of our system, to automate some document processing operations with Word and Excel. Most programmers used to calling COM from C++ should be impressed how much easier it is to do from Tcl!

Web services API's. Another common approach to integration between separate components is through some sort of network API, usually a web-based protocol, either simple CGI or something higher level like SOAP or XMLRPC. We'd in fact used this approach in the past to do some loose integration with some of our customers. Again, Tcl provided us with very compelling solutions. On the client side, the 'http' package provides a simple way to call a web-based API. The ease of embedding TclHttpd (or minimalist homegrown variants) and customizing them to provide an API, easily handles the server side. Today, there are also Tcl extensions that will ease supporting protocols such as SOAP.

This HTTP API was the approach taken when we eventually did the system integration between the audio and data conferencing systems, and it worked very smoothly. Tcl, as a string-based command language, is readily adept at implementing all sorts of network API's.

System structure. System structure obviously plays a large role in integration. Well-defined API's are easier to provide if the code is structured into clean modules. Localizing design decisions in modules reduces the impact of changing aspects of the system. For example, system-wide database calls were isolated in a few modules, rather than spread throughout the code, making database swapping a modest endeavor. Clearly, such system integrations need to be not only possible, but also achievable in reasonable time frames.

7.2. Configurability

Larger systems often need to be highly configurable, whether deploying different features sets for different customers or product versions, adding custom extensions for certain users (perhaps for additional integration needs), user interface branding, etc. Again, Tcl shines in this area.

Configuration language. It is no surprise that using Tcl as a configuration and extension language for the system can be a big win. Not only does this save the code to write a new parser, but also provides a more flexible configuration environment.

Dynamic vs. static. Because Tcl scripts are interpreted at run-time rather than statically compiled, making appearance changes, enabling or disabling features, and other such changes are straightforward to make. We often found it useful to isolate these sorts of decisions (e.g. appearance settings, or feature activation flags) in separate modules to simplify program structure.

Again, we had used this approach to implement simple branding and feature selection for previous customers. And because our application had gone through many iterations, we had more than a few aspects of the system that could be tweaked with run-time parameters. (It is possible to go too far of course, and several times we chose to go back and remove some of the older choices, which were adding too many cases to the code).

7.3. Reliability

System reliability, both robustness and fault tolerance, is critical for larger systems. We engineered reliability into our Tcl applications using the following techniques.

AOLserver. We relied on AOLserver for our central web server. Suffice it to say that it is pretty easy to make the case for AOLserver's reliability, given they day-to-day production use within AOL.

Script application features. Scripts don't core dump, unlike C code. Most day-to-day coding changes, such as fixing bugs and adding features were isolated in the Tcl script layers, not in the C code modules, which provided a lower level "engine" that stayed fairly stable. Any programming errors in the Tcl side were easily caught and logged so they can be fixed, while the program continues running. Regularly modifying C code is likely to introduce errors that can produce large side effects or system crashes. The more higher level code, the easier it is to prevent crashes in the face of errors.

Debugging and introspection. Various forms of logging, interactive debugging and introspection of system state are all significantly easier and more dynamic in scripting languages like Tcl than in more static languages, making it easier to find and eliminate errors.

Multiple process model. We divided our application into several different processes (e.g. web server, central database manager, one process per meeting room). Tcl makes this easy because of features like exec, and the ease of communicating between the processes using sockets, pipes or other IPC mechanisms. Splitting the system into multiple processes ensures a fault in one doesn't take down the system. Processes monitor each other and are restarted as needed. Contrast this with systems that make sharing information between components harder, necessitating a single process model (or a complex and error-prone threaded solution) where a single error can be fatal.

Development processes. Of course, reliable development processes such as automated builds, tests, code reviews, etc. are also critical to ensure system reliability, in any language!

7.4. Performance and scalability

Performance can be a general issue with large systems, but because we were developing a server application that needed to support large numbers of users, the scalability of the system was a very important and difficult issue.

This was complicated because with the Proxy Tk strategy we implemented, much of our client code, only ever designed to run on a single user's machine by itself, was now being run on the server, one instance per logged in user. Suffice it to say that it's a good thing we'd begun tackling this problem long before we started due diligence discussions!

AOLserver. Again, "standing on the shoulders of giants" gave us a great boost of credibility right off the top when it came to scalability. Not only were we using the same web server as was used to run one of the highest traffic sites in the world, but many of the same techniques and technologies (e.g. Tcl) that made AOLserver work so well were embedded in other parts of our system as well.

Monitoring performance. The key to any performance or scalability improvement is profiling and other measurement. This can tend to be a laborious exercise, and Tcl is probably no better or worse than any other language in this regard (though not having to recompile when we wanted to add more telemetry information was nice). Some of our coding style choices (e.g. single entry point for each module) did make it somewhat easier to add profiling information when needed.

When we did find areas that needed improvement (either in terms of the time that an operation took, or memory use by a part of the code), it was just a matter of slogging through and coming up with better ways to do things.

Scalability test harness. The other aspect of profiling, along with measurement in the server, was generating the needed load on the server. We developed a scalability harness (in Tcl) that simulated the behaviors of large number of Proxy Tk clients connecting to and banging on the server. Modifying the behavior of these clients was easy, because the protocol is simple strings sent over the network. Again, writing the scalability harness in Tcl rather than using another solution meant that it was one less tool our developers needed to learn.

Migration to C. One of the common solutions was of course migrating very time critical Tcl code into C. Over time, we moved a substantial portion of our "core collaboration engine" code (networking, data sharing) into C, leaving almost all application features in Tcl. Our coding style (structuring each module's public interface as a single command) again made this an obvious and low impact transition to make when required, and demonstrated that future performance improvements could be made in the same manner.

Multiple processes. The multiple process model helped us in a number of ways in regards to scalability. In our system, we tended to have multiple meeting rooms open at the same time, each having perhaps 10-20 users in them. Each meeting room had its own process, using the Tcl event loop to arbitrate between the 10-20 attached users with very little overhead.

Contrast this with a typical threaded solution with one thread per user, and the operating system overhead of managing the threads would quickly grow as the number of users and meetings increased). Relying on a non-threaded solution also of course helped simplify the solution and increase overall reliability.

Splitting the application into multiple processes also makes it trivial to scale the application to take advantage of multi-processor servers. Threading solutions of course would also have this same benefit. Unlike with threading, because the different processes communicated with each other via network sockets, it would also be possible (albeit with some code changes) to spread the processes over several different machines.

Excessive memory use is unfortunately one large drawback of scripting languages like Tcl, and our application not only used huge amounts of memory, but our code also tended to leak lots of memory over time. Using the multiple process model meant we could simply shut down the meeting room process when a meeting was completed and free up all that memory. Using a long-running process, though certainly possible, would have generated an endless amount of work.

Results. Our application was at one point measured at handling approximately 1000 simultaneous users, at typical activity levels, on a single very modest server, with reasonable load and low latency of operations. That's about 2500 separate Tcl interpreters!

While it did take us a considerable amount of measurement, tool development, tuning, recoding, and optimization, it does demonstrate that using Tcl as a development language does not prevent building high performing scalable systems.

8. Looking Back, Looking Forward

This section provides some general thoughts, both on our choice of Tcl leading up to and beyond TeamWave's acquisition, some potential issues that may hinder Tcl adoption, and some overall conclusions.

8.1. Why Tcl worked for us

It's clear that tackling the range of systems described here, from early academic prototypes through several robust and scalable commercial systems, would not have been feasible without Tcl. It provided opportunities to take this work significantly farther on much fewer resources than would have been required with more conventional development tools and technologies.

While in the early days, the natural fit of Tcl/Tk for prototyping work and small applications made it an obvious choice, this work suggests that Tcl also scales up well to "real" applications, when used appropriately. Development process definitely counts.

8.2. Potential Roadblocks

Tcl isn't the clear or easy choice for everything. It is still not a comfortable, mainstream choice (and probably never will be).

It's true that we did prove our case for using Tcl, and had very positive experiences as our team and technology merged into the acquiring company, and as new development continued. However, even many months later, there continued to be some degree of underlying suspicion and anti-Tcl bias.

It's also true that pitching Tcl for a new project would likely be substantially harder than in our case, where we were talking about adopting an existing body of code. Many of our arguments were convincing because we could point first-hand to the actual working system. Brand new systems would require arguing via anecdotes and second hand examples, which may be equally valid, but are less compelling.

Our experiences suggest that beyond general suspicion, there are a few potential roadblocks that could continue to impact future Tcl adoption within larger corporate applications, particularly in the area of web applications.

Interactive web applications. Unfortunately, for applications like ours that require very interactive web interfaces (more than is possible with forms or Javascript), there are few choices available. While the Tcl plugin still exists, and may be appropriate for some users, for mainstream use the download is an obstacle. The Proxy Tk solution we built was a good solution, but is proprietary and hence unavailable for general use. It would be nice to see new options along those lines available for these types of systems.

Web site development. For conventional web programming, Perl and PHP are still much more prevalent; Apache modules for using Tcl do not appear as popular or refined, and Tcl is not commonly used under IIS. TclHttpd does serve an important and distinctive niche, but it is not a mainstream solution. AOLserver is a fantastic product, with high credibility, but the latest versions are now available for Unix only. The removal of Windows support is a severe blow against corporate use, even if the main advantage was merely to say "yes, it runs on Windows if we ever need it to."

The need for best practices. There is a tremendous volume of invaluable information about Tcl, in FAQ's, Tcl-URL, the Wiki and many other excellent volunteer maintained resources.

One problem I see though is that there is not an easy-to-use, central resource truly focused on best tools and best practices, emphasizing a small set of mainstream tools and techniques used by a large set of developers. It is easy to find there are 23 different OO extensions; it is harder to learn that [incr Tcl] is probably the right choice if you want a stable and mature extension to add conventional OO structures to your application.

Without this, it becomes more important that a project using Tcl have a very strong champion or guru to stay aware of the current state of the language, monitor newsgroup announcements, know where the different resources are located, etc. Without a good focused resource, you almost have to become fully immersed in the Tcl community to become proficient. This is one factor that I think may be significantly hindering Tcl language adoption.

8.3. Summary

Certainly in my first experiments with Tcl, Tk, and Tcl-DP, I'd have never predicted that I'd be able to use the same basic set of tools to develop large, robust, production-quality collaborative applications supporting hundreds of users on a server.

For my work, Tcl has proven to be amazingly versatile. It lowered barriers to entry for novice programmers, and made it possible for individuals or small groups to attack much larger problems than they were able to before. Yet the same tool was also powerful enough to handle the more specialized needs of teams of professional developers on significantly larger and more complex projects. That is an amazing range for one tool.

It was good to find that with a bit of know-how, the mistakes that were encountered scaling up to larger applications are no different with Tcl than with any other language. Luckily, standard software engineering solutions can be applied fabulously well in Tcl.

Combining sensible software development practices with the huge productivity gains coming from a rapid development tool made using Tcl for our larger applications a very solid choice.

Acknowledgements

This work draws on the experiences of dozens of developers who have been involved in these projects over the years. Ted O'Grady brought in many of the key ideas and was instrumental in their implementation. Other key contributors include Brad Johnson, Raymond Yip, and Leo Pelland. This work has also benefited greatly from discussions with Jean-Claude Wippler. Ted O'Grady, Raymond Yip, and Larry Virden provided helpful comments on early drafts of this paper.

References

[1] Cockburn, Alistair. *Agile Software Development*. Addison-Wesley, 2001.

[2] Grady, Steven, Madhusudan, G.S. and Sugiyama, Marc. QuaSR: A Large-Scale Automated, Distributed Testing Environment. *Proceedings of the 1996 Tcl/Tk Workshop*. Monterey, CA. 1996.

[3] Johnson, Ray. Tcl Style Guide. http://www.tcl.tk/doc/. 1997.

[4] Libes, Don. Writing CGI scripts in Tcl. *Proceedings of the 1996 Tcl/Tk Workshop.* Monterey, CA. 1996.

[5] Libes, Don. Writing a Tcl Extension in Only Seven Years. *Proceedings of the 1997 Tcl/Tk Workshop*. Boston, MA. 1997.

[6] McCarthy, Jim. *Dynamics of Software Development.* Microsoft Press, 1995.

[7] McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 1993.

[8] Ousterhout, John. Tcl/Tk Engineering Manual. http://www.tcl.tk/doc/. 1994.

[9] Phelps, Thomas A. Two Years with TkMan: Lessons and Innovations. *Proceedings of the 1995 Tcl/Tk Workshop*. Toronto, Canada. 1995.

[10] Roseman, Mark. Tcl/Tk as a Basis for Groupware. *Proceedings of the 1993 Tcl/Tk Workshop*. Berkeley, CA. 1993.

[11] Roseman, Mark. Managing Complexity in TeamRooms, a Tcl-Based Internet Groupware Application. *Proceedings of the 1996 Tcl/Tk Workshop.* Monterey, CA. 1996.

[12] Roseman, Mark. Proxy Tk: A Java applet user interface toolkit for Tcl. *Proceedings of the 2000 Tcl/Tk Conference.* Austin, TX. 2000.

[13] Roseman, Mark and Greenberg, Saul. *Building Groupware with GroupKit*. In Harrison, Mark (ed). *Tcl/Tk Tools*. O'Reilly. 1997.

[14] Wippler, Jean-Claude. MetaKit. http://www.equi4.com/metakit/