Caubweb: Detaching the Web with Tcl

John R. LoVerso and Murray S. Mazer
The Open Group Research Institute
Cambridge, MA

Caubweb: Detaching the Web with Tcl

John R. Lo Verso and Murray S. Mazer

The Open Group Research Institute
Eleven Cambridge Center, Cambridge MA 02142 USA
{j.loverso,m.mazer}@opengroup.org

Abstract

Caubweb™ is a research system that allows a user to create local collections of Web documents on the user's computer, for access to those collections when disconnected. The system is part of a project investigating ways to provide adaptive, ongoing read and update interaction with Web-based information, even under conditions of variable or intermittent network connectivity. Caubweb is architecturally an HTTP proxy augmented with value-adding capabilities. To accommodate our design principles of platform-portability and extensibility, we used Tcl as our implementation language. This paper reports on our experience in using Tcl/Tk to build Caubweb. We discuss the structure of our implementation, identify strengths and weaknesses of the language and its tools, contrast Tcl/Tk with alternatives, and present a "call to arms" for the Tcl/Tk community, to promote increased reuse and cooperation.

Keywords: World Wide Web, Detachable Webs, HTTP proxy servers, Tcl, Tk, Disconnected operation, Mobility.

1. Introduction

Caubweb™ is a research system for investigating ways to provide adaptive, ongoing read and update interaction with Web-based information, even under conditions of variable or intermittent network connectivity. Caubweb is part of our group's Distributed Clients project [17], which has the broad goal of increasing the availability and customization of Web-based information services for mobile computing users. The expected benefits include increasing the availability of information, reducing the latency of servicing requests, and adapting information to the specific user and context.

Caubweb currently focuses on support for disconnected operation, implementing caching of user-specified "weblets" for access when disconnected. Caubweb also demonstrates the staging of user changes to stored documents when disconnected and integration of those changes into origin servers upon reconnection. These features are analogous to support for disconnected operation in file systems. In addition, a prototype visualizer named CaubView (which uses library components from Caubweb) provides views of the relationships among elements in a Caubweb cache.

Caubweb is architecturally an HTTP proxy augmented with appropriate value-adding capabilities. This accommodates our design principle of *vendor neutrality*, meaning that the functionality is not restricted for use with one vendor's browser or server. This is increasingly important as Microsoft, Netscape, and others work on increasingly incompatible technology. As identified by Brooks et al. [2], vendor neutrality can be achieved in many cases by adding application-specific capabilities to HTTP proxies, which can transparently filter, transform, and otherwise process the stream of HTTP requests and responses generated by the user's browser and the Web's origin servers[22].

To accommodate our design principles of *platform-portability* and *extensibility*, we chose Tcl[19] as our implementation language. This paper reports on our experience in using Tcl/Tk to build Caubweb. We discuss the structure of our implementation, identify strengths and weaknesses of the language and its tools, contrast Tcl/Tk with alternatives, and present a "call to arms" for the Tcl/Tk community, to promote increased reuse and cooperation.

Section 2 motivates the need for information access under variable connectivity conditions. Section 3 discusses the decision to use Tcl. Section 4 discusses the structure of the Caubweb application and its substantial use of library components. Section 5 presents an evaluation of Tcl and its toolset as we experienced it in building Caubweb. Section 6 revisits the decision to use Tcl. Section 7 summarizes our findings and urges greater cooperation and interaction within the Tcl community, in order to promote increased use of Tcl as "programming for the Internet" takes on greater urgency.

2. The Detachable Web

The World Wide Web has revolutionized information access, dramatically broadening the set of users and tasks for which network-based publishing and information access has become commonplace. We now think nothing of clicking on a hyperlink that points halfway around the world to retrieve even trivial bits of information.

Traditionally, users have been constrained to accessing information resources while in designated workspaces, such as offices or homes. They are increasingly able to access these resources elsewhere, with the increased availability of portable computers and wireless and remote communication systems[13]. Nonetheless, there will be many times when the user cannot contact remote information resources, and the user must make do with the data available from the local machine (in a mode called *decoupled computing*: the ability to compute when detached from the existing computing and communications infrastructure[11]). In this narrower context, the goal of the work reported here is to provide the user of a portable computer with ongoing interaction with Web-based resources when disconnected from the network infrastructure. We call this a *Detachable Web*. Our approach applies equally well to portable and "non-portable" user machines; the key technical challenge is to cope with periods of disconnection.

The intersection of these two trends (hyperlinked multimedia documents and portable computers) offers new problems not found in either setting. For example, work in disconnected file systems did not consider support "above" the file system level and did not consider embedded object references and access to associated services (e.g., an annotation service). Likewise, the Web community has only partially examined issues such as variable bandwidth fetching and presentation, providing Web-based services while disconnected, and merging documents created or modified off-line with available on-line servers.

2.1 The Approach Taken

The essence of the approach described here is: allow the user to specify *weblets* (connected subsets of Web content) of interest; cache those weblets locally; and, when disconnected, impersonate the servers on which the cached information resides. (This last aspect is achieved by trapping the requests for those URLs and serving them out of the cache.) In addition, if the user changes a locally stored document and publishes it toward its origin server, maintain the new and previous versions while disconnected; upon reconnection, integrate the new versions back to supportive origin servers.

The system described here does not depend upon changes in or specializations to Web browsers or servers.† Caubweb, as a proxy, is placed in the middle between the browser and the server. Therefore it can catch and divert user requests appropriately, acting for the most part transparently to the user, browser, and servers. Our specialized proxy provides caching, change staging and integration, weblet specification and retrieval, presentation, proxy configuration, and control.

The proxy approach has independently been pursued by commercial "off-line browsing" software vendors (e.g., WebEx[28]), who focus on read-only access and Windows-based platforms. Browser vendors are integrating off-line browsing support more tightly with their products, and other approaches use independent applications that take advantage of browser APIs for tracking activity and request display (e.g., WebWhacker[6])

Detachable Web support is analogous to support for disconnected file access. The primary target platform is portable client machines which experience alternating intervals of connectivity and disconnectivity, both voluntary and involuntary. The minimal goal of the system is to permit read-only access to a detached web. A next step is to support modifications to the detached web and to integrate those changes into the appropriate Web servers upon reconnection. As in some disconnected file systems[14], we assume we cannot make changes to the implementation of servers (but can use existing interfaces).

The primary differences between a system for Detachable Webs and a system for disconnected file access relate to ways of discovering object references to be cached, coherence requirements, underlying object model, and the semantics of object collections. For example, systems providing disconnected file access do not make caching decisions based on references to other files contained in already cached files or external services. A key notion in a Detachable Web system is following embedded hyperlinks, images or objects in a web page being cached or viewed. Other relevant object references may come from external services (such as PICS servers)[16].

†. Hence the acronym **Caubweb:** ***Caubweb*** *A*ugments *U*ser *B*ehavior *W*ith *E*very *B*rowser

2.2 Motivation for a Proxy-based Approach

One goal was to build a system that could work with any existing off-the-shelf browser or HTTP server. This allows us to avoid the problems that arise from modifying a browser, even if we could do so. Modifying a browser limits the new functionality to be available in a specific browser version or forces the developers to race to keep up with the ever-burgeoning number of proprietary browsers and servers. Further, it is no longer reasonable to assume that one can modify the browser the user wants to use. For similar reasons, we did not want to modify any features of the underlying operating system or network framework.

The kind of support Caubweb needs to provide is best represented as middleware. Consequently, we chose to base our system on the notion of *application-specific stream transducers*[2]. A stream transducer performs some specific value-added function for the user, usually transparently. As a stream transducer between the browsers and the servers, Caubweb can catch and divert user requests appropriately, allowing us to retain the use of unmodified HTTP for a communication medium between our component and other components.

If the world were not full of huge, monolithic clients with many features hardcoded into the application, then there would be more ideal ways to add Detachable Web support. In particular, if the user-side caching model of a browser were implemented against a simple API in such a way that it was replacable when the browser was deployed, then the caching engine for our system could be used to replace it. Such a modular approach does exist for some browsers (such as Internet Explorer), but not for the bulk of the off-the-shelf browsers with which we wish to interoperate.

3. The Decision to Use Tcl

Three principles guided our choice of implementation language: platform-portability, extensibility, and ease of distribution. Platform-portability means that the system should be usable on multiple platforms with as much code re-use across platforms as possible (or, equivalently, as little platform-specialized code as possible). Extensibility means that one can extend the capabilities and functionality of the software easily, through modular software interfaces, without appealing to vendors, and without being forced to work within the constraints of restrictive licensing. Ease of distribution implies preparing as few executables as possible to accommodate the set of target platforms.

These criteria pointed toward an interpreted language, eliminating languages such as C, C++, and Visual Basic. An interpreted language allows the same code to work on different operating systems and machine architectures without any changes. Machine dependencies are hidden inside of the interpreter, yielding a high degree of portability. Interpreted languages typically provide powerful string processing capabilities, appropriate for dealing with HTML, and thereby avoiding issues of dynamic string allocation, growth, and reference management. Interpreted languages often promote rapid development (typically trading off application performance) [25]. The end-user need not compile the sources or download a platform-specialized distribution in order to use an application. Finally, interpreted languages often have graphical user interface modules that are portable across platforms, relieving the programmer of the details of different windowing systems.

The primary candidates were Java[7], Tcl, and Perl[29]. At the time of our evaluation, Java was a freshly released language with rapidly evolving language definition, program development support, runtime support, and portability. That instability recommended against Java. Perl has been used successfully by many projects, including some of our own [18]. Perl supports a reasonable object-oriented programming style, has an interpreter augmented with a byte-code compiler (improving performance), has a well-organized, cohesive, user-contributed library, and is easy to embed in other programs. Nonetheless, Perl was not generally available at that time on Windows and Macintosh platforms and lacked strong visual interface support.

These factors and our own strong experiences led us to select Tcl, which was an excellent choice for an initial implementation of the system. The primary reasons were its ease of use, its interpreted nature, its relative maturity, and, finally, its promised portability to all the major computing systems we targeted. Because of our previous experience with Tcl, we believed that the language is easy to learn and understand (permitting new team members to become productive quickly), generally allows for the creation of highly readable code (promoting reuse and transfer of code responsibility), and is immensely fun to use. Section 5 discusses in detail our perception of the strengths and weaknesses of Tcl, based on the implementation of Caubweb.

4. The Structure of Caubweb

This section describes both the Caubweb application and the library components that provide much of the core functionality. The library, *Cobweb*†, is intended to be general purpose and can support applications other than Caubweb. We first describe Caubweb's major pieces,

†. Acronym available upon request.

followed by descriptions of modules in Cobweb used to implement interesting features of Caubweb.

4.1 Caubweb

Caubweb is an implementation of a Detachable Web proxy. It typically runs as a stand-alone Tcl program, started separately from (and usually before) the user's browser. It listens and responds to HTTP proxy requests at a given port (usually 8088 with connections restricted to the local host†).

4.1.1 Usage Model

The intended usage model of the system is simple. The user first starts a personal copy of Caubweb on a laptop (or other computer); the user then configures a web browser to proxy through Caubweb. When the system is well-connected to the rest of the world (while at work, for instance), the user browses the web as normal. Caubweb, as a transparent "middleman" in the browsing activity, follows the user's actions and caches the results of the user's interactions. The user can direct Caubweb to apply a *weblet retrieval* to pre-fetch resources asynchronously, so that (to some depth) resources connected by embedded or related hyperlinks will also be available to the user later. Weblet retrieval can be explicit (the user provides both the starting URL and the retrieval criteria) or implicit (the user sets default retrieval criteria, which are applied to each URL the user requests). At some point, the user will shut the system down.

Later on, when the system is no longer connected to the Internet at large (on the airplane, for instance), the user can turn on the laptop and start Caubweb, indicating not to use the network. Caubweb will serve HTTP requests with resources available in its cache. It can also note when the user browses outside the range of the resources in the cache; these cache misses can then be used to start a new weblet retrieval the next time Caubweb is told the network is available.

The following subsections list Caubweb's major functional components.

4.1.2 Caching HTTP Proxy

Almost everything that Caubweb does (or can do) is enacted by HTTP flowing into or out of the system. As a caching HTTP proxy, Caubweb listens for incoming user requests and directs outgoing HTTP requests. However, there is not a one-to-one correspondence between the incoming and outgoing requests. Weblet pre-fetching results in Caubweb initiating its own outgoing HTTP requests. Some incoming user requests may be to Caubweb's control panel, which is original content in a portion of the URL space for which Caubweb acts purely as an HTTP server.

Almost every resource that Caubweb receives is saved in an extended HTTP cache. This cache will be the sole source of documents when Caubweb is disconnected from the network. The cache nominally follows the requirements for an HTTP cache as defined in the HTTP protocol, RFC 2068 [4]. To provide better off-line browsing ability, Caubweb caches *all* documents that pass through it, including those that a normal caching proxy would not be allowed to cache. The external behavior apparent by observing Caubweb when connected to the network will always be that of a compliant HTTP caching proxy, meaning that it will appropriately discard cached documents while connected. This behavior allows Caubweb to serve possibly stale documents when disconnected. This is likely to be exactly what the user wants, since stale data is better than no data (as long as the user is provided with appropriate cues about the data's freshness).

4.1.3 User Interface

Caubweb's normal interface to allow the user to interact with and control the system is a *control panel* provided by a set of dynamically created HTML pages. These pages allow the user to view the overall status of the system, change various preferences and settings, and get listings of (and control over) the cache contents. The various changeable controls are implemented via HTML forms, with a hyperlinked help system for cues on use of individual controls. The control panel is accessed at a special URL, `http://caubweb/`, which uses a fictitious host served internally by Caubweb.

Caubweb provides an additional, optional user interface component that is implemented using Tk. This interface provides a hierarchical status display showing the state of events occurring inside the system. The user can gain, at a quick glance, information about Caubweb's activity at any time. The status display also includes some simple controls that complement those available on the control panel. More sophisticated user interfaces are certainly possible, but this was not the focus of our work.

The HTML control panel is the primary interface, and the Tk display is optional for two reasons. First, Tk is not required for the core functionality. Caubweb works as well using tclsh as it does using wish. Requiring the use of Tk to invoke control features would mean that Caubweb could not act transparently in the background, without taking up valuable screen real estate (as is usually the problem with many Windows programs). If

†. Except on systems lacking the ability (MacOS).

Tk is not required, then Caubweb can even operate without access to a display.

Second, while the overhead of the status display is not large when using Tk with the X Window System, it is about twice as compute intensive under Windows and on MacOS. Because of this, the status display can be disabled by an option or user preference, and the system simply ignores all the status panel code when Tk is not available.

4.1.4 Weblets

A weblet is a logical collection of documents that fit user-defined criteria. Individual documents may be part of many weblets at the same time, and weblets may overlap in content. Weblet members are identified dynamically as needed. As a result, at any instant in time, it may be impossible to answer "what resources are in this weblet" because all the possible members are not known. Weblet membership is ephemeral - Caubweb does not currently keep persistent membership information.

A weblet is normally identified by some starting point (typically a URL) from which all other documents in the weblet are related. The relationship to the starting point is established by a *weblet template* which is a collection of *qualification predicates* a document needs to meet. A weblet template is applied first to the starting point of the weblet, and then, recursively, to any additional documents matched by the template. Each qualified document is parsed into a list of its internal hyperlink references, such as anchors, images, frames, etc. A document is added to the cache by virtue of this weblet if it is (1) referenced in a document already in the weblet and (2) matches the qualification predicate.

The *qualification predicate* is one of several forms of test that may be applied to a document or its properties, such as the URL. These tests include:

- a pattern match on the URL (e.g., http://www.opengroup.org/RI/*)
- a pattern match on the anchor associated with the hyperlink (e.g., "Research Institute*")
- a depth limit (e.g., no more than three links away from the start URL)
- tests against other properties (e.g., size, type, etc.)

Predicates are conjoined and disjoined to form the expression in the weblet template.

The Caubweb control panel has several convenient ways for the user to create new weblets. The simplest is a small form that handles most of the typical cases via a set of pre-defined templates. A more complicated form allows finer control, while an expert form allows arbitrary weblet templates to be specified in the weblet language.

The user can apply a template to a start URL to start a *weblet retrieval*. This causes Caubweb to start a background task fetching the documents that comprise the weblet. This allows Caubweb to cache documents that the user may never have visited.

Weblet retrievals may also be started automatically via a default weblet template. When this is enabled, Caubweb will apply the template to every URL the user's browser requests. When the user utilizes this mechanism for normal browsing, the result is a cache that is "rich around the edges" with content (possibly) related to something the user was interested in.

4.2 The Cobweb Library

As mentioned earlier, our library is intended to be highly reusable and serve multiple purposes. Cobweb contains several major subsystems, including:

- an asynchronous execution model based upon events and callbacks
- an object-oriented execution environment, based on `obTcl` [5]
- several Web-centric modules (URL, HTML, HTTP, caching, weblet, etc.)

In addition, there is a collection of other useful modules (e.g., host name resolver, splash screen, and *comm* facility for implementing Tk send(n) over sockets).

Cobweb is intended to be architecture-neutral. However, there are several places where it needed to be cognizant of the host system, usually to work around limitations or bugs on the Mac or Windows ports (see Section 5.1).

Highlights of the library are described below, with examples of use in Caubweb.

4.2.1 Asynchronous Execution

Most of the library follows an asynchronous, non-blocking model of execution. This model follows from the event-driven nature of Tk and Tcl's fileevent (see also Section 6.1). Applications such as Caubweb will have many partial operations in progress at any one time. This model means that each ongoing operation will get a chance to accomplish some work when it is able. However, each operation must gracefully relinquish control after doing some small amount of work.

Cobweb requires any method that is not short-lived and non-blocking to use either a *callback* or a *continuation* to guarantee liveliness of the application. A callback is a small script that a caller passes to a method, after which the caller is expected to relinquish control. Upon completion of the request, the callback has a return value appended and is then evaluated. This results in control being returned to the caller. The callback will occur in the context of the facility to which the caller made the request, meaning that the caller cannot assume anything about its own state. An example use of callback in Cobweb is to initiate an asynchronous request to an HTTP server and later receive a handle for the connection.

A continuation is used by a component to schedule its own, long running operation. A continuation is similar to a callback but adds some state that the caller passes in and then later receives as part of the upcall. An example use of continuations is to maintain the state of a weblet retrieval while other operations gain and relinquish control.

4.2.2 Web-centric Modules

URL: this is an obTcl class that can manipulate URL syntax. It can parse a URL string into component parts, as well as reconstruct it with a `tostring` operation that allows the use of a *base* URL object. This is used to resolve relative URLs.

HTTP Protocol: this is an obTcl class with a simple interface and several stackable implementations. Http uses several structures to track individual connections. These structures include the message (msg), connection (conn) and HTTP header (header). Built into the module is the logic for making outgoing requests directly or via a proxy.

The basic interface includes methods such as MakeReq (which uses a callback) and Close. Implementors of the interface are modules that an application will invoke, causing the modules to initialize themselves into a protocol graph. For instance, if an application just initializes the Http and HttpProxy modules along with a server loop, the result will be an HTTP tunnel (that is, a cacheless proxy). If it also initializes the HttpCache module, then it becomes a caching proxy.

HTTP Server: This includes a server loop and a simple, *mock* HTTP server. The server loop enables a socket listening on a port or ports to process incoming requests. Each request is broken apart into command and target and then dispatched to the handler for server using that socket**.** The HTTP server reads disk files and returns their contents to the remote side. It understands several special CGI-like file types. One is called *htcl*, which is Tcl code to be evaluated in a slave interpreter, the output of which is sent to the remote. Another is *thtml*, which is an HTML file with embedded Tcl code that is expanded via subst. *htcl* is used to create the Caubweb control panel pages, which include dynamically updated data from Caubweb's state.

HttpCache: This is an obTcl class that configures itself *above* Http in order to transparently trap requests from the higher levels and take appropriate action based on the state of the cache.

HTML Parser: This is a variant of the HTML parser from sntl [23] combined with changes from SurfIt [1], which are based upon Steve Uhler's html_library.tcl[30]. It has been heavily modified so that it is re-entrant and can parse incrementally. Additional work went into cleanly splitting the main logic apart so that it is no longer driven to render the HTML it parses. Finally, it can be told to parse against tagsets, so that a document can have particular information located in it. Caubweb uses a tagset to find rendering elements (e.g., <IMG>), hyperlinks (e.g., <A HREF>) and other assorted references.

Weblet: This module implements the "weblet walker," which is responsible for identifying and fetching into the cache all members of each weblet. It makes substantial use of work list and queue management abilities.

5. Tcl's Strengths and Weaknesses

Arguments about the suitability of Tcl for any given project usually focus on several key concepts, such as the interpreted nature, the "everything is a string" model, its use as a glue language, and the ease of constructing user interfaces with Tk. These aspects derive directly from Tcl's roots but do not necessarily reflect the use of the language to implement complete stand-alone applications such as ours. We used Tcl to construct an entire application, which allowed us to focus on some different key aspects of the language. We present some of our observations here.

5.1 Portability

Tcl, being a scripting language, affords a natural portability of code. This concept, along with the (at that time) forthcoming Windows and Mac ports of Tcl 7.5, meant that an implementation of our system in Tcl could easily allow us to "buy into" a truly portable system. This mostly came true. To date, we have run Caubweb snapshots on Windows95 and Windows NT, most flavors of UNIX, and the Macintosh (System 7.0 and higher). In addition, the InfoPad project at the University of California, Berkeley, has used Caubweb on its base station running Unix to provide disconnected access to web pages via the InfoPad. However, support on multiple platforms required more effort than we originally expected.

Creating pure Tcl is a useful approach to avoiding machine dependencies. It means that end users can utilize scripts without work such as compiling new code. Prior to Tcl 7.5, the general rule was that a new extension had to compile its modules and then statically link with the Tcl libraries itself. This process resulted in large binaries with limited mixtures of extensions. The new **load** command resolves this problem.

However, not everything can be made with pure Tcl. Caubweb depends upon two key mechanisms (*copychan* and *host*) that are implemented in C as a minor extension. This requires us to provide source code and force end-users to compile it. We take the middle approach of also providing the pre-compiled loadable modules for some number of *supported* platforms. This still requires work to compile the module on all the platforms we support, but that happens infrequently, when we make a change in the source of the loadable module.

This work is often well worth doing. We have long believed that some Tcl/Tk applications win users over better than others based upon the convenience of installing and using the application. When the system is portable with little or no effort, a user will be inclined to use it. The Tcl community is littered with systems that have not achieved broad user acceptance, such as *tkwww* [27] and *tknews* [26]. On the other hand, systems such as *exmh* [31] match this model.

The negative side effect of using a compiled extension is that it becomes an additional portability constraint. Requiring a mechanism that is only available as a compiled extension (or some other compiled C code) adds an obstacle for the casually interested party to try out the system on an unsupported platform. This may well reduce the set of people who choose to evaluate a new system.

This trade-off was constantly evaluated during the project to decide when and what components were to be used in the system. In the end, we ended up with three loadable modules, only one of which was required by our system (copychan). The other two (*host* and the obTcl accelerator) are optional; pure Tcl code exists to implement the functionality of each module if it is not available. This allows us to get the performance gain of the loadable module when we have done the compilation work, allowing us to speed up the system when we are committed to it on a particular platform.

This trade-off was also partially responsible for us not using other extensions, like MTtcl [10] or Incr Tcl [15], although other reasons played a role for these extensions (i.e., they were not initially available for Windows and the Mac).

Even with Tcl's high degree of portability, we still had significant problems. Our first concern was (at that time) the ongoing development of Tcl 7.5. The Mac and Windows ports contained many bugs that prevented the system from being directly used. For much of 1996, we had to provide our users with bug fixes for Tcl 7.5 and later Tcl 7.6. To avoid causing our end users to acquire the tools to compile under Windows and MacOS, we also provided binaries of the fixed Tcl/Tk distribution on those platforms.

5.2 Extensibility for Performance

For Tcl, the extensibility mechanism is a direct trade-off against portability. But, when performance really matters, it is time for a compiled C extension. In the end, we took a cautious approach of only utilizing loadable modules that were either absolutely required for Caubweb to work or for which we could easily provide a workaround that was written in plain Tcl (at a performance penalty).

We added *copychan* for this reason. It is a derivation of *unsupported0*, which has become *fcopy* in Tcl 8.0. However, it has two important characteristics. First, it properly returns the number of bytes copied and, thus, never loses data. This allows us to use it reliably to copy data from a web server into a cache file and know that the cache file is correct (fcopy also has this characteristic). Second, it allows us to "multicast" the data to multiple file channels. This allows us to utilize the same data stream from a web server to create the cache file and to return the data to the user's browser. The result is that we save significant compute time doing a read/write/write from a Tcl loop. In addition, prior to Tcl 8.0, it was impossible to program that loop in Tcl at all, since there was no reliable way of storing the file data in a Tcl variable (since the data could contain arbitrary binary data).†

One area in which additional performance would have been useful was in parsing and manipulating HTML. Until the ability was added to parse HTML files piecemeal (using a trick from SurfIt! [1]), this was a major cause of perceived sluggishness in Caubweb. However, when weblet retrievals are in progress, the overhead of parsing HTML consumes the vast majority of Caubweb's compute time. Caching the hyperlink references derived from the HTML parsing may alleviate this problem.

†. Several extensions would have helped in this regard, the most interesting being the memchan extension [12].

6. Revisiting the Decision to Use Tcl

Many of the basic reasons for choosing Tcl have paid off handsomely. The language has proved suitable for constructing our system. The ease of writing has proven itself over and over, as new capabilities are easily added to the system. The learning curve for producing usable, good code for project engineers has been short.

A significant advantage has been achieved in the lack of problems related to fixed size string buffers, data structures, memory allocation, garbage collection, and stray pointers. These common problems have plagued numerous systems, including other web systems (servers and browsers).

In addition, Tcl mechanisms sometimes lead to particularly elegant implementation techniques. The best example of this is the weblet template language, which is internalized into a form that can be directly evaluated by the Tcl interpreter. As a result, for very little cost we have created a powerful and expressive syntax for describing web page relationships without being burdened with building the expression evaluation engine.

Nonetheless, some aspects of Tcl were problematic; we discuss these below.

6.1 Issues in Event Handling

Tcl has an inherent event-driven nature. This is derived partially from the historic implementation of Tk and windowing systems, and due to bias on the part of the language designer [19]. This forces any complex system such as Caubweb to use an architecture of asynchronous execution with callbacks. This is a workable solution, but it does have drawbacks. Any operation that blocks the return to the event loop causes the application to appear to hang momentarily. There are some ways to avoid this, such as the work to make the HTML parser compute in smaller quanta. However, there are situations in which one cannot avoid blocking and for which Tcl provides no help. Hostname (DNS) lookups in the socket channel code is one example. This causes the application to become entirely unresponsive, and the user to complain. This is true for Caubweb and for other Tcl-based systems (this is a common user complaint about exmh, which even has a secondary helper process to avoid this problem).

An alternative to this approach might be to use a threaded operating system with a threads extension to Tcl and then partition the application so that significant work is done in separate interpreters, each in their own thread. This is the approach take with Audience1 [21]. By having operating system support for threads, a single thread can block in an operation and not prevent other threads from continuing execution. This naturally limits the use of the application to those systems that support this collection of abilities. Further, the model of separate interpreters causes other problems, as state (variables) can not be shared across interpreters and thus must be duplicated, with the cost being paid in the memory footprint of the system.

To contrast, even a language like Java designed with threads in mind only partially escapes this problem. Threads are supported even on systems where there is no underlying operating system support. Java forces all I/O to be asynchronous but does not force the event-driven model on the programmer. Java actually goes the other way; whereas I/O is asynchronous, the I/O interfaces are not. Thus, to read on a stream, a thread is forced to block.

6.2 Tcl Changes

Our work began when Tcl 7.5 was still in alpha testing. Our initial framework used TclX 7.4 as a stopgap until 7.5 was stable enough. The process of working against a moving target has occasionally been painful. Not having a Windows port of Incr Tcl meant that any useful object-oriented extension needed to be pure Tcl so that we could use it on all our platforms. Along the way we have helped to debug several serious problems in the socket code for the Windows and Mac ports. It was not until Tcl 7.6p2 that we finally were able to stop shipping our own Tcl binaries.

This is not a complaint about the work that the Sun Tcl Group produces, but rather an observation about differing priorities. Their priorities have been different than what we would have liked. Completeness and correctness in the socket code is more interesting, *to us*, than completeness in native look and feel.

6.3 The Impact of Tcl 8.0

The worst shortcoming we have come across in our Tcl work can be summed up in two quick points: the need for extensions to support basic language operations (OO and megawidget paradigms, binary I/O) and the overall performance of an interpreted system. These have been identified by many people in the past [21]; as of this writing, Tcl 8.0 has been released in beta form, and a new age is upon us.

To be sure, it does not solve all the problems. However, performance measurements show some marked improvements in some areas, new I/O mechanisms will allow arbitrary data to be manipulated without the potential loss of information, and a common core mechanism for namespaces opens the door to a unified OO mechanism.

These potential benefits come with a price. Our system will need substantial changes to work with the new incompatibilities introduced with these new mechanisms. For instance, obTcl uses the familiar **"::"** separator between class and instance, but this is now usurped by the namespace facility. Consequently, none of our existing obTcl-based code can work with Tcl 8.0b1. With obTcl now no longer being maintained, we must revisit the decision to use obTcl. Will Incr Tcl be adapted to the new namespace mechanism quickly?

In addition, while *fcopy* replaces some of the mechanism of *copychan*, it does not handle multiple destinations. Hence, there will still be a need for our own modified versions.

6.4 Call to Arms for the Tcl Community

When reviewing features that different languages have to offer, one notices that Tcl is missing a rich, cohesive, organized, standard library. The community, while building many fine components and libraries, does not have a mechanism in place to collect those components in a single library structure. Tcl code is too often reinvented, ignoring the advantages that comes from reusing existing software [24].

The Tcl contributed archive is a start at constructing such a library. However, much of the code in the archive is out of date and little of it can be expected to work together. The archive is exactly that, a static collection of what has been done rather than a unifying repository against which applications can be built.

In contrast, other language environments come with these components. Java has the JDK (and more). Perl has a rich set of primitives plus CPAN [3], a well organized library of additional code. CPAN is so successful that a single implementation of a module tends to be adopted by the community; further, changes by the community are often actively merged back into the library.

We believe that Tcl needs an archive like CPAN. The existing contributed Tcl archive includes reusable code, but the cost of reuse is almost always too much work. A single, cohesive library would mean there would not be five different HTML parsers or four different HTTP implementations, and that Tcl developers could create new and interesting things rather than spending time reinventing these same old pieces.

Tcl, to date, has avoided providing much in the way of auxiliary library code. Tcl 8.0 breaks from that tradition in a significant fashion by providing an new *http* package that implements some of the most useful client-side HTTP operations. This could be the start of a new, useful extended library, if only other existing useful code is incorporated.

Additionally, the core needs to embrace other sorely missing operations as Tcl language constructs. We believe there is a need for lower level bindings to a complete set of POSIX C library functions. This mechanism would support *scripts* in accomplishing work that currently requires a loadable module written in C.

When considering such a community library, one also needs to consider what might be expected from the Tcl core to empower reusable code. With the support of namespaces, an encapsulation mechanism for the benefits of data hiding can become standard. A full object-oriented environment is not necessary, but a mechanism like Incr Tcl's *ensemble* or the *major/minor* extension alone might be sufficient. Other noteworthy items include the framework changes for megawidget support and the core hooks to provide useful debugger support (such as single stepping).

6.5 Contributed Core Changes

The evolution of the Tcl core has progressed too slowly, in contrast to Java where change happens too quickly. This forces the Tcl developer community to create needed infrastructure themselves. If the core eventually incorporates this mechanism, it typically uses an incompatible implementation. By taking the slow approach to incorporate these changes, the effect of making the developer's code out of date is that the community always has new hurdles to overcome just to keep current with Tcl.

7. Conclusion

We have developed a substantial server-based application, targeted to multiple platforms, using Tcl. We selected Tcl for platform-portability, extensibility, and ease of distribution, and it satisfied those criteria relatively well. The extensibility turned out to be particularly important because of deficiencies in evolving versions of Tcl that were either platform-specific or related to core functionality. We identified strengths and weaknesses of Tcl that stood out during our work, and we challenged the Tcl community to work together to create a rich, cohesive, standard library.

The Caubweb application uses a set of components (Cobweb) which we have already re-used in an ancillary application called CaubView. It is based upon HistoryGraph [8][9], the result of another project in our organization that used Tcl. HistoryGraph was a visualizer that allowed a user to dynamically view relationships among web pages retrieved by the user's browser. A revised version of HistoryGraph serves as a prototype for viewing relationships within the Caubweb cache. Revisions included adding new relationship calculations and replacing some prototype components with more robust equivalents from Cobweb, such as the HTML parser, the URL class, and the HTTP protocol modules. Incorporating the Cobweb components took less than one week.

Availability

The latest version of Caubweb (including the Cobweb library), instructions for its use, release notes, and user mailing lists are available for our web page:

http://www.opengroup.org/RI/www/dist_client/caubweb/

Caubweb is covered by distribution terms that allow free use and redistribution of the system or Tcl source for non-commercial research and evaluation purposes.

Acknowledgments

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under the contract number F19628-95-C-0042. The views and conclusion contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Stavros Macrakis identified some of the initial important concepts in the Distributed Clients project. Charlie Brooks developed the merge-back aspects and substantially improved CaubView, the initial version of which was contributed by Scott Meeks, based on the HistoryGraph work of Meeks, Frederick Hirsch, Charlie Brooks, and Murray Mazer [8]. Charlie Brooks, Frederick Hirsch, Susan LoVerso, Mary Ellen Zurko, and the workshop reviewers provided useful feedback that improved the contents and presentation.

References

[1] S. Ball, "SurfIt! A WWW Browser," *Proc. Fourth Annual Tcl/Tk Workshop,* Monterey, CA, USA, July 1996, pp. 161-171.

[2] C. Brooks, M.S. Mazer, S. Meeks, and J. Miller, "Application-Specific Proxy Servers as HTTP Stream Transducers," *Proc. Fourth International World Wide Web Conference*, Boston, MA, USA, December 1995, pp. 539-548. http://www.w3.org/pub/Conferences/WWW4/Papers/56/

[3] Comprehensive Perl Archive Network (CPAN), http://www.perl.org/CPAN/CPAN.html

[4] R. Fielding et al., "Hypertext Transfer Protocol - HTTP/1.1," RFC 2068.

[5] P. Floding, patrik@dynas.se, obTcl 0.57b3, ftp://ftp.dynas.se/pub/tcl/

[6] Forefront Technologies. http://www.ffg.com/whacker/whacker_tech.html

[7] J. Gosling, B. Joy, and G. Steele. *The Java^{TM} Language Specification*, Addison-Wesley, 1996.

[8] F.J. Hirsch, W.S. Meeks, and C.L. Brooks, "Creating Custom Graphical Web Views Based on User Browsing History," *Sixth International World Wide Web Conference*, Santa Clara, CA, USA, April 1997. http://www.opengroup.org/RI/www/waiba/papers/www6/hg.html

[9] F. J. Hirsch, "Building a Graphical Web History Using Tcl/Tk," Fifth Tcl/Tk Workshop, Boston, MA, USA, July 1997.

[10] S. Jankowski, booga@netcom.com MTtcl documentation, ftp://www.neosoft.com/pub/tcl/sorted/devel/MTtcl1.0.tar.gz

[11] R.H. Katz, "Adaptation and Mobility in Wireless Information Systems," *IEEE Personal Communications*, 1 (First Quarter 1994), pp 6-17.

[12] A. Kupries, a.kupries@westend.com, "Memory Channels in Tcl," ftp://ftp.westend.com/pub/aku/memchan1.2.tar.gz

[13] M.S. Mazer et al. "Issues in Mobile Computing Systems: Guest Editor's Note," *IEEE Personal Communications, Special Issue on Mobile Computing*, December 1995, pp. 12-13.

[14] M.S. Mazer and J.J. Tardo, "A Client-side-only Approach to Disconnected File Access," *Proc. IEEE Workshop on Mobile Computing Systems and Applications*, December 1994, pp 104-110.

[15] M. J. McLeannan, "[incr Tcl]: Object-Oriented Programming in Tcl," Proc. of the Tcl/Tk Workshop, University of California at Berkeley, CA, June 10-11, 1993.

[16] J. Miller, P. Resnick, and D. Singer. "Rating Services and Rating Systems (and Their Machine-Readable Descriptions)," *World Wide Web Journal*, Vol. 1, No. 4, Fall 1996, O'Reilly and Associates, Inc.

[17] The Open Group Research Institute, Project Overview for Distributed Clients, http://www.opengroup.org/RI/PubProjPgs/dist_clients.html

[18] The Open Group Research Institute, Project Overview for Intelligent Browsing Assistance for the WWW, http://www.opengroup.org/RI/PubProjPgs/waiba.html

[19] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[20] J. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)", *Proc. 1996 USENIX Technical Conference*, January 1996.

[21] A. Sah, et al., "Programming the Internet from the Server-Side with Tcl and Audience1^{TM}," *Proc. Fourth Annual Tcl/Tk Workshop,* Monterey, CA, USA, July 1996, pp. 183-188.

[22] M. Schickler, M.S. Mazer and C. Brooks, "Pan-Browser Support for Annotations and Other Meta-Information on the World Wide Web," *Proc. Fifth International World Wide Web Conference*, Paris, France, May 1996. http://

www5conf.inria.fr/fich_html/papers/P15/Overview.html

[23]S. Shen, slshen@lbl.gov, Sam's New Tcl Library 0.4, ftp://www.neosoft.com/pub/tcl/sorted/devel/sntl-0.4.2.tar.gz

[24]H. Spencer, "How to Steal Code -or- Inventing The Wheel Only Once", *Proc. Usenix Conference Winter 1988,* Dallas, TX, USA, February 1988, pp. 335-345.

[25]T.H. Romer et al., "The Structure and Performance of Interpreters," *Proc. ASPLOS VII*, Cambridge MA USA, October 1996.

[26]tknews news reader, version 1.2b, ftp://www.neosoft.com/pub/tcl/sorted/net/tknews.1.2b/tknews.1.2b.tar.gz

[27]tkWWW World Wide Web browser, version 0.12, http://www.mit.edu:8001/afs/athena.mit.edu/course/other/cdsdev/html/welcome.html

[28]Traveling Software, http://www.gowebex.com/

[29]L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl (2nd Edition)*, O'Reilly and Associates, Inc., 1996.

[30]B. Welch, S. Uhler., "Tcl/Tk HTML Tools," *Proc. Fourth Annual Tcl/Tk Workshop,* Monterey, CA, USA, July 1996, pp. 172-182.

[31]B. Welch, exmh mail reader, http://www.smli.com/~bwelch/exmh/