The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

Programming the Internet from the Server-Side with Tcl and Audience1

Adam Sah, Kevin Brown, and Eric Brewer
Inktomi Corporation

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Programming the Internet from the Server-Side with Tcl and Audience1™

Adam Sah, Kevin Brown and Eric Brewer
asah@{inktomi.com, cs.berkeley.edu}
kebrown@{inktomi.com, haas.berkeley.edu}
brewer@{inktomi.com, cs.berkeley.edu}
Inktomi Corporation
2168 Shattuck Ave. Suite 210, Berkeley, CA 94704

Abstract

Although innovations on the World Wide Web are currently dominated by exciting client-side products (ie. Java, VRML, Netscape Plug-Ins, etc.), we believe that there is an equally rich server-side programming opportunity. In this paper, we argue that server-side programming will remain an important part of client-server Web applications. We then argue that server language(s) and client-side languages have very different requirements, and don't have to be the same language.

As a running example, we present Audience1™, an end-to-end publishing tool for the World Wide Web, which uses Tcl and MTtcl, a multi-threaded Tcl extension library. Currently, Audience1 is providing web service and mass customization features for the HotBot search engine (a joint venture between Inktomi and HotWired). HotBot can be found at `http://www.hotbot.com`.

I. Introduction

The World Wide Web is a general-purpose, distributed, client-server computing environment. Very high level languages and toolkits play very different roles in each.

On the client side, it is handy to provide tools tuned to the tasks of user interface (UI) design and implementation, security, network communication, installation, configuration, and uninstallation. Thus, the current client-side computing environments all offer a rich set of products to fill these needs, some bundled with the operating system, some as third-party packages and some as both. HTML, Plug-Ins, Java and VRML are rushing to provide these services on the Web.

The server side looks much like the client side, except that user interfaces are less important, and access to sophisticated multi-user data management (ie. DBMSs), report generation, publishing tools and scalability become the dominating issues. Inktomi, BroadVision [Broa95], NeXT [NeXT95], Navisoft [Nav95] and other companies are beginning to develop Web server products that fill various niches of this market. Both Inktomi's and Navisoft's products are based on Tcl.

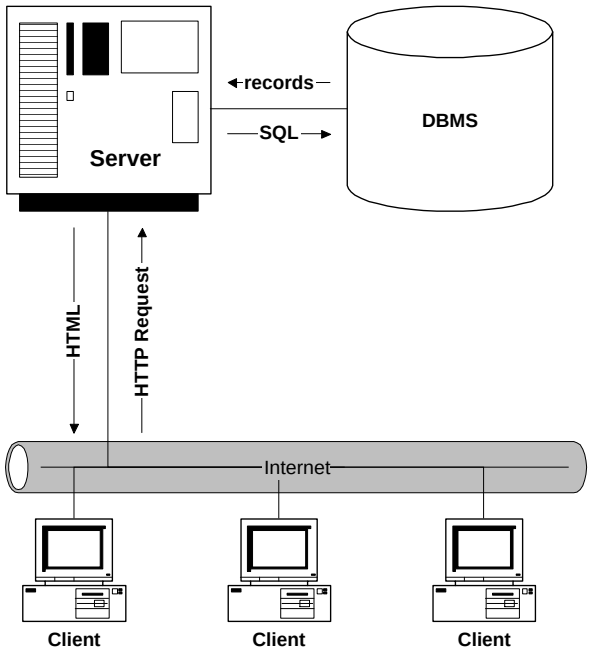

Figure 1. Client-Server computing on the Web

The remainder of this paper discusses issues in the design and implementation of a sophisticated, scaleable dynamic Web server. In the next section, we review how World Wide Web servers work, why dynamic HTML generation is important and why the Common Gateway Interface (CGI) doesn't scale well. In section

3, we explore the value of server-side programming in greater depth and introduce Audience1. In section 4, we describe the design and implementation issues of Audience1's scripting language, Dynamic Tags™. In section 5, we describe what features of Tcl Dynamic Tags stresses and what backward-compatible suggestions we have for language improvements. Finally, section 6 presents our conclusions.

II. An Overview of Client-Server Computing on the World Wide Web

All Web servers operate over the HyperText Transfer Protocol (HTTP), which dictates how clients and servers communicate. The HyperText Markup Language is one of the data formats supported by HTTP; by convention, all Web browser software (ie. Netscape Mozilla, NCSA Mosaic, Microsoft Internet Explorer) provide HTTP client service and automatic HTML formatting.

First generation Web servers are very similar to file servers: an HTTP request is made on a well-known network port, and a file is sent in reply (data) along with a few simple headers (metadata). In this model, the content delivered to clients is static, which means that all clients are presented the same data, and a given client is given the same data on repeated accesses.

Second generation web servers add the Common Gateway Interface (CGI). Instead of responding to HTTP requests with static content, a program is executed. The output of the program is used as the HTTP response. Since the response is still HTML, but can be different on every request (eg. by including the number of visitors to the site on the page itself), we call this "dynamic HTML generation."

Once in place, there are numerous uses for dynamic content generation:

- content targeting, such as custom newspapers.
- targeted marketing and advertising.
- Web server administration and online publishing tools, customized to the content being edited.
- "do the right thing" presentation: don't offer services that the user can't use or doesn't need. For example, don't present a page containing a Java script for a user not able to execute Java programs.
- "do the better thing" presentation: allow the user access to all services, but make the more useful ones easier to access. For example, in presenting a list of services, put the ones that the individual user is more likely to want earlier in the list.

As a means for producing dynamic content, CGI is flexible, easy to use, and compatible with virtually every programming language. CGI has the downside that it is slow and scales poorly because it requires a (heavyweight) process to be spawned for each Web request. Popular Web sites can receive hundreds of requests per second (millions per day); examples include Intel's home page (http://www.intel.com), stock quote servers, and several search engines.

An alternative to CGI is to bind the language used to create the dynamic content directly into the Web server itself. Although one can use C or C++ for this purpose, the performance of these languages is overkill and their programming models are cumbersome and lead to less robust web services: a scripting language is a better fit.

III. Dynamic Web Service and DBMSs

Dynamic HTML generation is incredibly powerful, but requires some additional features to realize its potential:

user identification. On the Web, users are not only anonymous but are identified only by IP address: it's important to distinguish one user from another so that the Web publisher can tailor or target content to users whose prior behavior or expressed preferences suggest his/her interests. For example, it's useful to ask "what percentage of our users use feature X?".

In Audience1, we include user IDs in the URL and/or a cookie [Nets96]. When a user first visits the site, a unique number is assigned to her, and a version of the site's home page is dynamically generated for her. Later visits automatically present her user ID to the system, although the URL-base scheme requires that she use a bookmark. For example,
`http://www.hotbot.com/UI28736487g` is the URL that user ID "UI28736487g" might use to revisit the site in the future. Bookmarks will include this additional information, and the UID won't interfere with form submissions.

browser targeting. The plethora of Web browsers has left Web publishers in a quandary: if they provide content that all users can see, they cannot use the latest features and have to revert to the least common denominator. If they employ the latest features, they risk alienating the users with legacy browsers.

However, if we know the capabilities of the user's browser, we can produce the best quality content that the browser can handle. For example, JPEG images are generally smaller than GIF images and therefore load

faster, but not all browsers can view JPEG images. With browser independence, we can provide the GIF version for less-capable browsers. As another example, many legacy browsers do not support cookies, which Audience1 uses for user identification; for these browsers, Audience1 automatically embeds the user ID in the URL.

It is easy to discover the browser capabilities because the HTTP request already contains the name of the browser, which can be correlated with a database of browser capabilities. Some capabilities may need to be provided manually by the user, such as size or color depth of the client display.

dynamic HTML rewriting. Many of Audience1's features require specialization of the HTML being sent to the client. For example, Audience1 can rewrite inline image tags to use more advanced image formats if the browser supports them. If the publisher specified foo.gif, but also provided foo.jpg and foo.pjpg, and the client browser supports Progressive JPEG, then foo.pjpg will automatically be specified in the HTML for the page, a rewrite of <img src=…/foo.gif> ➔ <img src=…/foo.pjpg>. Rewriting is also used for transmitting user ID information in the URL for hyperlinks pointing to the same site.

persistent storage. Since user-specific data needs to maintained over a long period of time, and can be difficult to reconstruct if lost, the obvious solution is to store this data in a database management system (DBMS), thus requiring DBMS access to be bound into the Web server as well.

With user identification, client browser identification and DBMS access, we can bring Web services into the mainstream of corporate communication. Unlike paper mail, telephones and faxes other communication media, the Web provides a bi-directional channel, where both channels are computerized and lossless: a company can produce content and receive feedback on that content instantly. Feedback can either be explicit (such as a form the user fills out) or observed (such as which pages are most popular among different user groups). DBMSs provide fast query capability, allowing the company to react to that feedback instantly.

For example:

- If the web server stores logs hits in a DBMS, log analysis can be performed "on the fly". We can target advertising to users based on their behavior.
- If we keep track of a user's preferred language, we can present content in that language, if available.
- We can keep track of a user's user-interface customizations, automatically using these as the defaults during his/her next visit to the site.
- Frequent visitors to the site can be offered coupons and other specials. For commerce sites, frequent buyers can be targeted for special offers.

IV. Dynamic Tags, DBMS Access, and Publishing

Unfortunately, it is not enough to provide user IDs, browser identification, and persistent storage in the abstract. For these facilities to live up to the promises we have outlined we need additional facilities:

database access. Although most DBMS vendors provide pleasant front-end tools for designing, managing, and accessing databases, we need programmatic support from the scripting language. This involves communication issues, parsing issues, data representation issues and memory management issues. These details are beyond the scope of this paper.

Web-based report generation and data access tools. Access must be provided to manage the basic Web data schema (hits, users, pages, etc.). All of these services should be made available through an HTML interface.

schema- and language- extension. It must be straightforward for Web publishers to augment the database to track additional data items. For example, it is common to present a site-specific form to a user when the user creates an account on the site. Since this form contains site-specific fields, the database, report generators and scripting language must all provide facilities to manage this data.

migration from static publishing. It must be possible to migrate an existing site into using the tools of a dynamic Web server. In Audience1, Dynamic Tags™ allow publishers to add small amounts of dynamicism to existing HTML pages, gradually adding more dynamic elements to otherwise-static pages. Unlike the simple dynamic tags that other systems allow, Audience1 tags provide powerful features that hide large, complex add-on services.

Dynamic Tags™ look like HTML tags with an "@" sign prepended to them; unlike normal HTML tags, Dynamic Tags™ are parsed on the server; their output is normal HTML, sent to the browser for further processing. In the context of Java, JavaScript, and other client-side languages, Dynamic Tags™ allow servers to provide the dynamic inputs to the client-side scripts. Java doesn't replace the need for server-side processing because you

can't download most databases to clients, and even if you could, there are privacy and security reasons to avoid it. It is also infeasible to transmit large data sets (eg. MRI scans of the user) to the client: users will rarely view more a small portion of large datasets, and so it is more efficient to cull the data set on the server before communicating the result.

One example of Dynamic Tags™ is <@ad>, which places an advertisement on the page at the position indicated by the tag. <@ad -keywords "hard drive"> might find an ad targeted to customers interested in hard drives. Ad targeting is a black art, where good results contribute directly to one's bottom line revenue as more customers "click through" the ad. Therefore, good ad targeting software can consume a large engineering effort spent developing expert heuristics. <@ad> and its arguments hide this complexity from the publisher.

The architecture of Audience1 is shown in figure 2. Although pages are served just like NCSA httpd and other traditional web servers, the HTML is intercepted before it is served to the client, and the Dynamic Tags converted to static HTML (executed). Although you have to trust the author of the web server software, we still try to provide a "safe" Tcl interpreter in case a rogue script causes undue harm or is coerced into causing harm by the client's HTTP request. Figure 3 shows an example page.

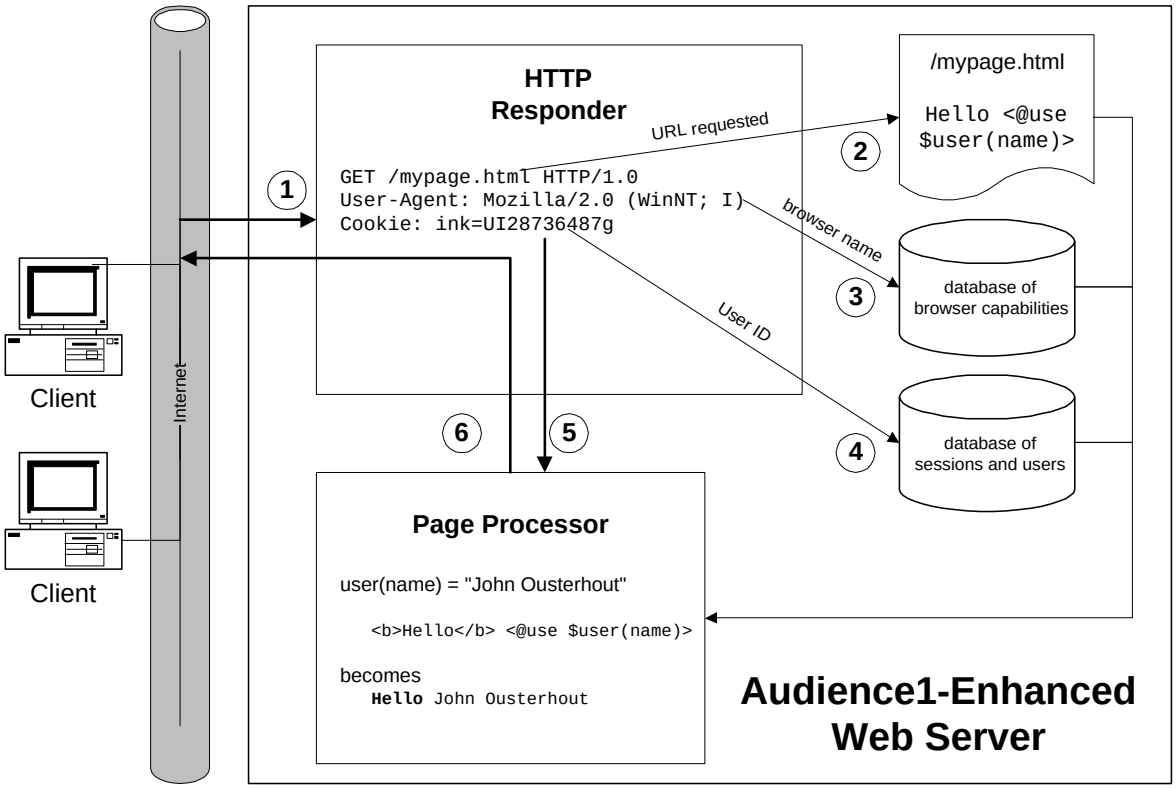

Figure 2. The Audience1 architecture.

(1) At the core of Audience1 is a HTTP responder that listens for incoming requests and assigns a thread to handle each one. (2) Each request is accompanied by a Uniform Resource Locator (URL) request, specifying which (dynamic) HTML page to access, along with a user ID. If the page is valid, the page source text is loaded. (3) The responder provides the User-Agent header (the name of the browser), the client IP address and other HTTP header information to the page processor, processed to be as useful as possible (ie. browser name becomes browser capabilities, IP address becomes location information, such as country). (4) From the user ID, Audience1 links (via a DBMS JOIN) to the user's personal and preference data, if available. Note that user IDs are anonymous unless the end user explicitly attaches it to personal data. (5) This data is then made available to the page processor, whose job it is to sift the HTML for Dynamic Tags™. For each dynamic tag, the processor executes the Tcl code associated with the tag. (6) The output is returned to the user.

```
Dynamic HTML is backward compatible with static HTML, so the following text
produces a standard HTML <title>:

   <title>This is the title of this example HTML page</title>

Anywhere on the page, you can place Dynamic Tags, which get evaluated at the
server.  The simplest such tag is "@use", which is like "puts", but prints the
output to the client browser.  For example, the following Dynamic HTML prints the
server machine name:

    Hi— the current page ID is <@use $server( hostname)>

Dynamic Tags provide a myriad of services, from  shorthands for generating static
HTML (e.g. @table, which generates HTML <table>s) to advanced HTTP features such
as reading/writing HTTP Cookies and form submission values, to  DBMS access.  The
following example combines all of these features in the "Add URL" page of HotBot,
which saves the requested URL to the relational DBMS for future processing.
Because we store the user ID with the URL, we can reward frequent contributors
with coupons and other promotions.  Comments are interspersed in the left-hand
margin; we have left out the boilerplate static HTML.

Begin an HTML form, in which the "Add URL" text entry widget can appear.  The
current page is named addurl.html— i.e. the form is submitted back to itself.
This makes it easier to maintain the code because everything's in one place.
   <FORM action=/addurl.html>

Include a text entry widget named 'newurl' that's initially empty, and has room on
the screen for at least 40 characters.
   <@text newurl "" 40>

Next to the text entry field, include an image (… send.gif).  Arrange that clicking
on the image submits the form under the name "send".  End the HTML form.
   <@submitImage send send /hb/stuff/send.gif border=0>
   </form>

Audience1 allows authors to include raw Tcl code in their documents.  Here's the
code to process the form submission.
   <@tcl>
       # has the form been submitted yet? (or did the user just get here?)
       if {"$newurl"==""} {
            if [catch {
                 # save the URL in the DBMS in the table ' addurl', where
                 # the column 'url' is set to $newurl, and
                 # the column 'uid' is set to $user(id)
                 # the -save option tells the DBMS to overwrite an existing
                 # entry there.
                 @db_save -save addurl [list [list url $newurl]] [list [list uid
$user(id)]]
            } msg] {
                 # [error message printing code deleted for concision]
            } else {
                 @use "   $newurl...<p>   <font
size=+2>Got It! <font size=-1> Thanks, we'll come visit your site within a
week.</font>"
            }
       }
   </@tcl>
```

Figure 3. An example HTML page containing Dynamic Tags™.

The idea behind Dynamic HTML and Dynamic Tags is trivial: the real value-added to Audience1 are the Tags it provides and the quality of implementation behind each one. In the commercial system, Audience1 tags and their C libraries provide concurrent access to thread-safe services (such as the DBMS and the search engine), user-defined logging facilities, sophisticated debugging facilities, etc.

V. Tcl's strengths and weaknesses

It is often argued that Tcl's usefulness is tied to the Tk user interface toolkit, yet this example shows a counterexample. Unfortunately, Tcl is not the perfect tool for our purposes. This section explains the pros and cons as we've observed them.

Most of the pros are well-known to the Tcl community: interpretation, everything-is-a-string semantics, easy embedability in C, config files as Tcl scripts and so on. The unusual features we especially appreciated were:

thread safety under Solaris. Our search engine and web server are multi-threaded to allow us to hide the latency of I/O-bound services. This demands a thread-safe scripting language, which MTtcl [Jan95] provides.

lightweight interpreters. MTtcl creates one interpreter per thread; it is imperative that interpreters use a small footprint, which Tcl's do.

low performance risk. Unlike other languages, it is straightforward to extend Tcl from C. This reduces the risk that poor performance will affect product delivery. If a script is too slow, it can be rewritten in C. Note: this technique does not scale (see below).

variable traces. Variable traces on array variables allow us to provide users with the illusion that all accessible data has already been loaded out of the DBMS [Sah95]. In fact, loading data from the DBMS is a heavyweight operation that we would want to cache (memoize). With variable traces, we can load all related data once per HTML page that actually uses that data.

simple syntax. Anyone believing that "syntax doesn't matter" doesn't understand training costs. With Perl or C, it costs a company tens of thousands of dollars per employee in training and lost productivity. The basics of languages such as Scheme and Tcl can be taught to a new user in one day, plus another lost day of productivity fighting with the quoting rules. In developing HotBot, many of our engineers learned Tcl "on the fly" where none found similar success with Perl. Tcl's simple syntax made it feasible to implement Dynamic Tags as Tcl procedure calls. Such an implementation exposes Tcl's syntax to HTML page authors when they substitute variables and call subcommands in the arguments to a dynamic tag.

The following were the weaknesses in Tcl that most affected us:

it's slow. The Berkeley/Sun reference implementation of Tcl is very slow. For HotBot, this becomes a problem when Tcl script processing dominates the time it takes to return results to the end user, causing us to rewrite many procedures in C. If coding and debugging in C were a fraction as productive as in Tcl, we wouldn't need scripting languages; it is very painful to rewrite more than a few procedures in C. We didn't experiment with the Tcl-to-C compiler from ICEM/CFD; it is unreasonable to ask users to integrate a C compiler into their web publishing process, which would be needed to "compile" dynamic HTML pages statically.

no object support in the core. For both engineering reasons and marketing reasons, object-oriented programming (late binding, type inheritance, implementation inheritance, etc.) is here to stay. Books such as Design Patterns [GHJV95] argue that OOP provides an elegant platform for reasoning about code reuse, one of the most successful productivity improvement techniques we know of today [Bro95]. OOP needs to be in the core to be useful: without it, there's little hope that object-based extensions will become widespread. Namespaces (new in Tcl v7.5) are a good first step.

VI. Conclusions

Although the client side of Web programming has been won by Java, there is ample room to pick winners on the server side. We believe that Tcl makes an excellent candidate because of its availability on a plethora of platforms and the number of tools and extensions for it and its embedability.

References

[Bro95] Frederick P. Brooks. The Mythical Man Month: 20th anniversary edition (with new and recent essays). Addison-Wesley. 1995.

[Broa95] BroadVision Corp. *BroadVision marketing literature*. http://www.broadvision.com

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.

[Jan95] Steve Jankowski. *MTtcl documentation*. ftp.aud.alcatel.com:/tcl/extensions/MTtcl1.0.tar.gz

[Nav95] Navisoft Corp. *NaviServer documentation*. http://www.navisoft.com

[Nets96] Netscape Communications Corp. "Persistent Client State: HTTP Cookies". http://home.netscape.com/newsref/std/cookie_spec.html

[Next95] NeXT Corp. *WebObjects marketing literature*. http://www.next.com

[Sah95] Adam Sah. "*Multiple Trace Composition and Its Uses*," Proc. Tcl/Tk'95 Symposium. Toronto, Canada. June, 1995.