Crossroads
The ACM Student Magazine

An Architecture for Easy Web Page Updating

By ***John Aycock*** and ***Michael Levy***

Introduction

HTML is fast becoming ubiquitous at our institution, and we are not alone. Indeed, it is difficult to find a serious enterprise now that does not have a presence on the Web.

One of the reasons for HTML's success is its simplicity. Furthermore, there are a number of good tools available for web page creation. But when we took a close look at our site's web pages, we found that we live in a rather static neighborhood. Pages that should be changing were not being updated. Surely with powerful tools and a simple markup language, this should be an easy task! Why was our Web so static?

In software engineering, it is known that software maintenance can consume `over 70 percent of all effort expended' [**7**, p. 663]. We propose that the same holds true of many web pages, where frequent maintenance -- or updating -- is critical to their content: event calendars, course announcements, and product price lists are but a few examples. There is no reason to assume that tools for web page *creation* will permit easy updating.

The situation is even more complex when one considers the people performing the updates. They may not be the people who originally created the web pages. Their computer expertise may range from expert to novice. They may occupy a position in an organizational hierarchy, and may share updating responsibilities with others. They likely have little time or patience to learn to use yet another application. In this paper we propose an architecture for web page updating that addresses all these concerns.

The Myth Of The Atomic Document

Even the humblest home page is often a multi-topic affair, listing both professional and personal interests. A course web page will have both lecture and laboratory information.

These and other examples have led us to the conclusion that *a document is the fundamentally wrong level of abstraction for web page updating*. Updating requires a much finer-grained view of a web page; tools that deal with HTML on a page-at-a-time basis do not support this view.

Instead, we see a web page being composed of regions. A **region** is a block of text which may optionally have one or more regions embedded in it. A region also has some security attributes, which determine who is permitted to update the region. In many ways, regions are analogous to files and directories in a file system.

While this model sounds simple, it handles real-world updating situations. As an example, a course web page might be a single region generated by a system administrator, an HTML skeleton to ensure consistent formatting. Inside that region is another region owned by the course instructor, who can then create subregions and delegate them to laboratory instructors. Another example is a calendar of events, where each day is a separate region that any member of a group of office staff can modify.

The above examples suggest a nesting of regions that would yield a tree structure. In fact, we suggest that the region concept be further generalized to allow a directed acyclic graph structure. The model should allow one region to be embedded in several locations, both internal and external to a web page. Internally, one could have a region consisting of a horizontal rule or bullet graphic used throughout a document; simply modifying the one region would change it in all locations. Externally, one could easily maintain common headers and footers for a set of web pages by having them all refer to common regions. Judicious use of regions could even permit updating of web pages where attractive presentation is as important -- or more important -- than content.

Effectively what this model gives us is akin to schemes that allow objects to be embedded in documents, such as OLE [3] or the late OpenDoc [6]. The major difference is that we are only interested in blocks of text, not any application-specific or platform-specific ties the text may have.

Update Architecture

We now have a workable model of web pages, but we still need a means to implement it and address the differing needs of users.

Introducing a new application to perform web updates is an obvious first choice, but is almost certainly the wrong way to go. Novice users would need training; expert users can have ``religious'' preferences about their tools; pleasing both camps of users is difficult [9]. From an implementation perspective, there is already a large body of existing application code which can be reused.

There are other issues behind the scenes:

- Security. User authentication, as well as blocking or mitigating the effects of hostile actions such as replay attacks [11].
- Administration. Maintaining change histories; upgrading software and software configurations.
- Concurrency. Multiple users sharing data give the usual set of concurrent data access problems [10].
- Networking. No assumptions should be made as to what machine a user is updating a region from; all updates should be possible across the network. But this admits a new class of problems. For instance, what is the difference between a user taking a long time to update a region, and a user whose computer has been powered off?

Again, there is already code to address these problems in other contexts that is suitable for reuse.

Our architecture for web page updating is shown in **Figure 1**. A division is made between the **front end**, which is what the user sees, and the **back end** that manages updates and generates revised copies of HTML documents.

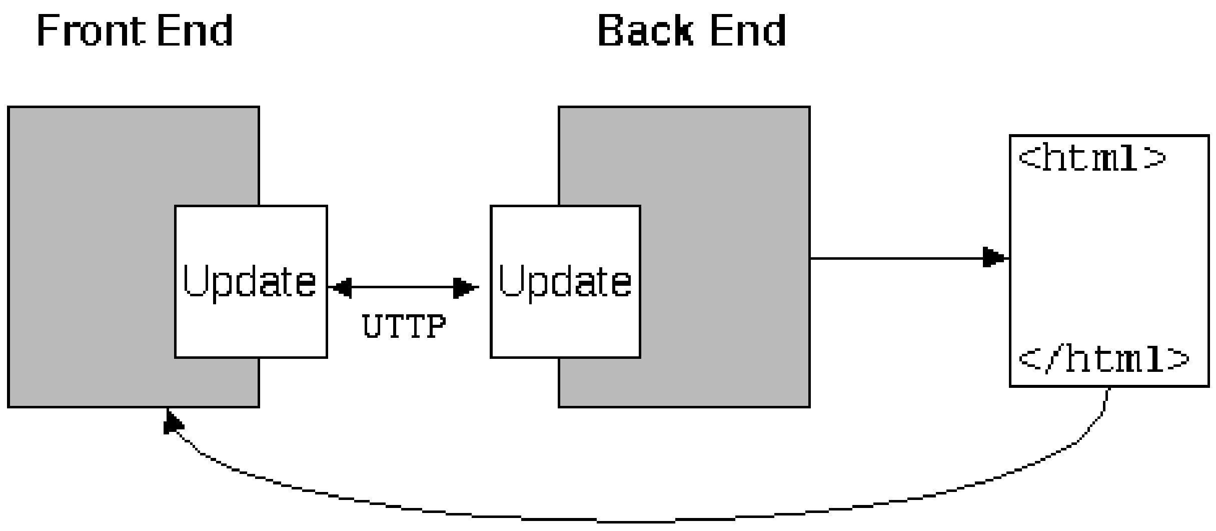


Figure 1. Update architecture overview.

An ideal front end is one that the user does not have to learn. Fortunately, most users are comfortable using at least one word processor or text editor. Modern word processors have web page creation support, and some text editors (like Emacs) are programmable. It makes sense from both the user interface and the code reuse points of view to create plug-ins for these applications to handle web page updating. One can extend this architecture to include a browser-based user interface for updates [2].

There are two parts to the back end. The first part must arbitrate concurrent updates and archive regions. The bulk of this task can be handled by using existing database software, or even revision control systems like CVS.

The second part of the back end is responsible for piecing regions together in such a way that an end-user's browser will render the web page properly. There are three approaches:

- The back end can produce the web page itself.
- The back end can make the HTTP server produce the web page, using a mechanism like server-side includes, where the HTTP server preprocesses web pages [5].
- The back end can attempt to make the end-user's browser produce the web page,

using HTML 4's <OBJECT> tags [13]. (This is the least reliable method since it makes assumptions about what tags browsers support.)

The front and back ends must communicate across the network. In light of the Document Object Model (DOM) recently becoming a Recommendation of the WWW Consortium, it makes sense to adapt it for this role -- DOM is an object-based API for manipulating documents [12]. Our own implementation predates this and uses UTTP, the UpdaTe Transfer Protocol [1], which will be described in the next section.

In contrast to other architectures, ours encompasses a wide variety of user interfaces, requires no modifications to the HTTP server, and provides for fine-grained web page updating and concurrency control.

Experience

Three years ago, we began to look at the problem of web page updating. As a proof-of-concept of our architecture, we have implemented a front and back end, gaining some valuable insights in the process.

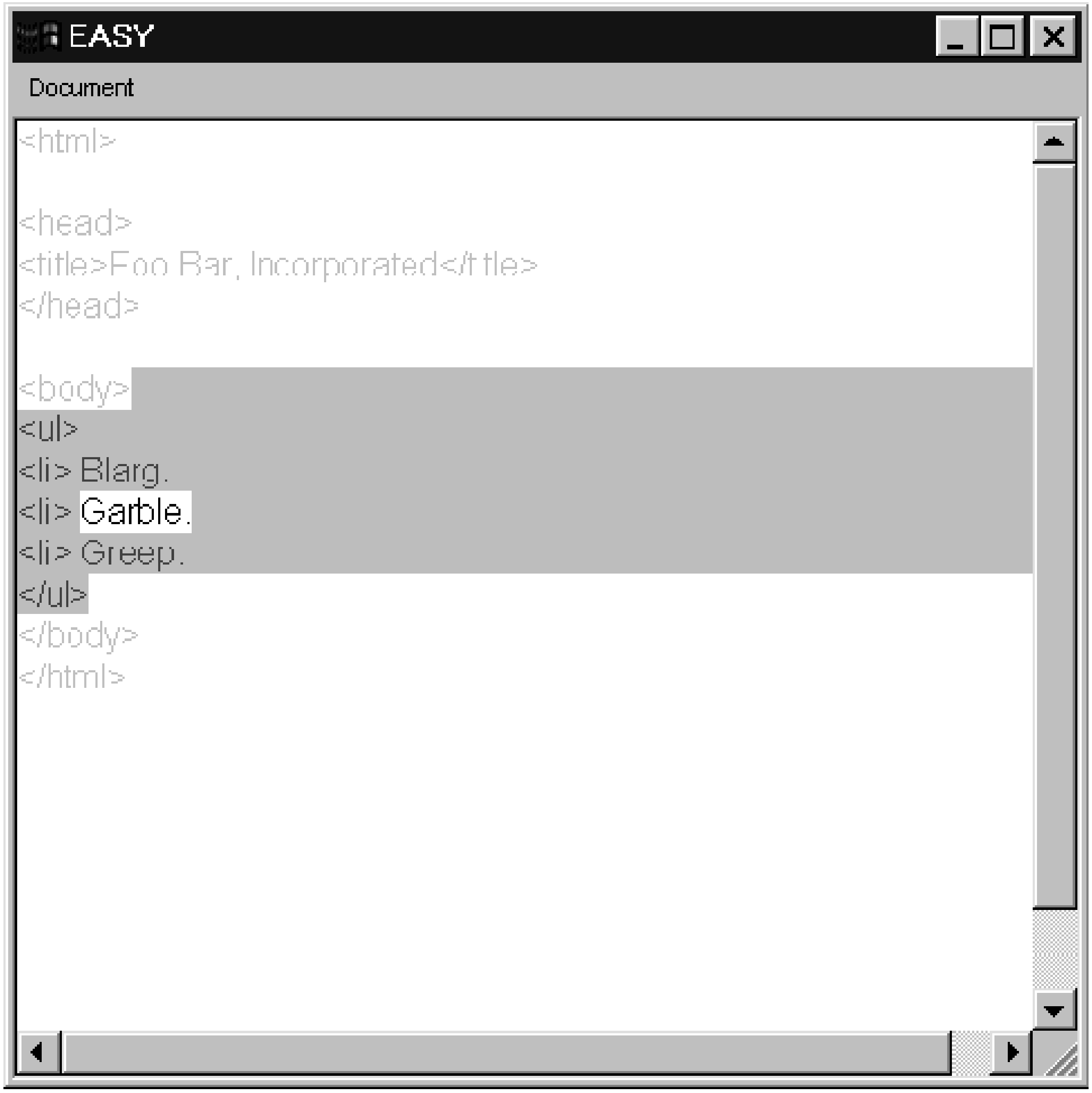
Front End

Although it seems contrary to the architecture presented in the last section, our front end is a stand-alone Java application, rather than a plug-in module for some existing program. In doing so, we could conduct user interface studies in ``neutral territory'' without having to consider prior experience with a particular application. (There is actually a Java class in the front end which handles UTTP transactions, so it does adhere to the architecture in some sense.)

The most difficult operation is selecting a region to update, as far as the user interface is concerned. There are four pieces of information that must be conveyed to the user:

- where the boundaries of a region are;
- what the structure, or nesting, of regions is;
- which regions the user has permission to update;
- what the ``current'' region is, or how the user selects a region to update.

Figure 2 is a screen dump from our front end, showing one way to convey this information. The text of regions that cannot be updated due to insufficient permission is rendered in light grey. As a user moves their mouse pointer over the document, the region underneath the pointer -- the current region -- is highlighted in light blue, delineating its boundaries. (In the figure, the highlighted region appears as dark grey rather than light blue.)



```
<html>

<head>
<title>Foo Bar, Incorporated</title>
</head>

<body>
<ul>
<li> Blarg.
<li> Garble.
<li> Greep.
</ul>
</body>
</html>
```

Figure 2. Selecting a region.

In our user study, this interface was well-received, and all but one person found the correct number of regions that they could update. When we later experimented with larger documents, we discovered that this does not scale well: a large, updateable region under the mouse pointer can turn the entire text window light blue, which is not particularly useful. An area of future research would be to find scalable ways to display the region information.

Back End

Our back end consists of approximately 2500 lines of C code. It uses GNU dbm databases to store the regions of a web page; the current version does not allow the sharing of regions between web pages. Once a region is updated, the back end directly produces a new HTML file.

Interestingly, the biggest problem with the back end was a matter of policy rather than a technical issue. In our zeal to appeal to novice users we had identified web pages not by their URL, but by a document name such as might be encountered in a regular file system. The problem: how does one map these ``simplified'' names into a location on a web site? Given the prevalence of URLs in popular media now, they are familiar to novices and still meaningful to experts -- no simplification was required!

UTTP

Like HTTP itself, UTTP is a text-based protocol. A client (the front end) establishes a TCP connection to a UTTP server (the back end), then repeatedly issues commands to the server and receives responses.

UTTP commands come in four flavors:

- Authorization. UTTP authorization is the same as that used in the POP3 protocol [**4**]. Each user has a secret -- a password -- shared with the server in advance. For every TCP connection, the UTTP server prints a banner message which contains a unique one-time identifier. A client authenticates a user by collecting the secret and the unique identifier, then computing an MD5 digest [**8**] on their concatenation. The client sends the user's name and the digest to the server, which can compute the same digest to verify authentication. Note that no secrets are sent in cleartext, and replay attacks are impossible because of the unique identifier.
- Information. These commands let a client ask questions like ``what documents contain regions that I can modify?'' or ``what are all the user names?'' (The latter is intended to help clients minimize errors when changing the permission attribute of a region.)
- Region management and updating. These commands are the *raison d' être* of UTTP, and it has a rich set of them. However, a client would typically only use the commands to read, lock, update, and unlock regions. TCP connections consume both network and OS resources. Since a user may take a relatively long time to edit a region, a UTTP client is allowed -- encouraged, in fact -- to drop the

connection to the server during this time. The client may reconnect at a later time with the slight penalty of having to re-authentiput /home/rathius/Documents/xrds/xrds6/xrds6-2/new/updatingnew.html /home/gglass/crossroads/xrds6-2/cate itself. A side effect of this feature is that a client may disconnect for an indefinite amount of time, holding the lock on a region so that no other client may modify it. For this reason, provisions are made in UTTP for a client to ``break'' an outstanding lock after a respectable period of time.

- Exception reporting. A sure way to aggravate a user would be to let them update a region, only to have their changes discarded because the region's lock had been broken by another user. UTTP does not try to avoid this situation, but it does provide a mechanism so that a user may receive timely notification of such events. A client may specify a **callback port**, a TCP port the client listens at. The UTTP server will announce exceptional events like lock breaking to these callback ports, so that a client can take appropriate action.

Provisions were made in UTTP to allow servers to garbage collect ``orphaned'' regions that no longer have any references to them. The concern was that unreferenced regions would unnecessarily consume resources on the server. The implication of this decision is that UTTP clients must issue some requests in a precise sequence. One cannot simply create a region, lest the server garbage collect it; one must update another region to have a reference to the not-yet-created region, *then* create the new region. The orphaning of a region is reported as an exceptional event to any registered callback ports.

Conclusion

Web page updates are falling by the wayside, even for pages where up-to-date content is vital. While web page creation tools are extremely powerful, their view of a document as an atomic entity is at odds with how people actually design and update web pages. By viewing a web page as the composition of regions, we can more accurately support the granularity needed for updating.

Combined with our architecture for updating web pages, this gives us a way to easily update web pages while minimizing the user learning curve and maximizing reuse of existing code. Perhaps most important in this fast-moving area is that our architecture is flexible enough to embrace new standards as they emerge.

Acknowledgments

Fang Lin developed an early prototype of the updating back end, and Shannon Jaeger

commented on a draft of this paper. The anonymous reviewers made several helpful suggestions.

References

1
Aycock, J. and Levy, M. UTTP Protocol Specification (April 1997). Unpublished memo.

2
Flypage Updater. **http://www.flypage.com/**.

3
Microsoft Corporation. OLE Documents: Technical Backgrounder (May 1994). Available at **http://www.microsoft.com/oledev/olemkt/oledoc/doctech.htm**.

4
Myers, J. and Rose, M. Post Office Protocol -- Version 3 (May 1996). *RFC 1939*.

5
National Center for Supercomputing Applications. Server Side Includes (SSI). Available at **http://hoohoo.ncsa.uiuc.edu/docs/tutorials/includes.html**.

6
Nelson, C. OpenDoc and Its Architecture. *AIXpert* (August 1995). Available at **http://www.developer.ibm.com/library/aixpert/aug95/aixpert_aug95_opendoc.html**.

7
Pressman, R. S. *Software Engineering: A Practitioner's Approach*, Third Edition. McGraw-Hill, 1992.

8
Rivest, R. The MD5 Message-Digest Algorithm (April 1992). *RFC 1321*.

9
Shneiderman, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Second Edition. Addison-Wesley, 1992.

10
Silberschatz, A. and Galvin, P. B. *Operating System Concepts*, Fifth Edition. Addison-Wesley, 1998.

11
Stallings, W. *Cryptography and Network Security: Principles and Practice*, Second Edition. Prentice Hall, 1999.

12
World Wide Web Consortium. Document Object Model Level 1 Specification, Version 1.0 (October 1998). Available at **http://www.w3.org/TR/REC-DOM-**

Level-1/.

13 World Wide Web Consortium. HTML 4.0 Specification (April 1998). Available at http://www.w3.org/TR/REC-html40/.

Biography

John Aycock (aycock@csc.uvic.ca) is a Ph.D. student at the Department of Computer Science, University of Victoria. His research interests include compilers and programming languages.

Michael Levy (levy@csc.uvic.ca) is an Associate Professor at the Department of Computer Science, University of Victoria. His research interests include web programming and middleware.