An Architecture for Integrating OODBs with WWW

Jack Jingshuang Yang and Gail E. Kaiser

Columbia University

CUCS-004-96

Abstract

The main topic of this paper is how to structure information so that the *view* of the web, both within and across web pages, is *dynamically customizable*. We present an architecture that integrates Object Oriented Databases with the World Wide Web to organize such dynamic structures. Different users, or the same user at different times, could have different views of the web. We discuss several architectural variants and and implementation issues. Our chosen architecture provides high flexibility for a wide variety of applications, ranging from managed healthcare to software development environments, and has been realized in the **dkweb** system.

Keywords

Views, Information Structuring, Meta Information, Embedded Methods

Introduction

The World Wide Web (WWW) is a huge, distributed information store. It has been very successful in providing information to users all over the world. The development of protocols such as the Common Gateway Interface (CGI), and technologies such as proxies, have enabled construction of numerous web-based applications, such as collaborative editors and annotation systems.

However, the information retrieved from the web is presented to users on an as-is basis without a "big picture" that associates the structure of the data with the specific application. In other words, users always view the web according to the low-level structure determined by hyperlinks within the web pages that contain the actual information.

A Web View is an abstract structure imposed on the web, rather than defined entirely within the web pages themselves. Different users might have different views of the same subweb; the same user might have different views at different times, appropriate to the task at hand.

We show through some motivating examples that dynamically customizable web views would be very useful. We present an architecture and an implementation that integrates object-oriented database (OODB) technology with the web to support views that can be dynamically customized for web-based application systems.

Motivating Examples

Say one physicist wants to use the experimental data posted on the web by another physicist, but in some different way. That is, the data is hyperlinked in a structure that matches the author's understanding of the logical relationships but the reader's perspective is different. The reader should then create a new view over the data to restructure and/or reformat the pages, for human browsing and/or input to computational tools.

Some healthcare systems use WWW to store patient information such as personal data, diagnoses, healthcare history, test results, physician notes, etc. Patients may be admitted to different hospitals within even a short period concerned with a single illness or injury, so logically centralized information may be distributed over multiple websites. Patient data should not be identically visible to all (authorized) users: administrative personnel don't need to see CT images, whereas medical researchers should mine histories and clinical trials anonymously. Thus information should be structured differently for each user role and/or application, while minimizing expensive duplication of information.

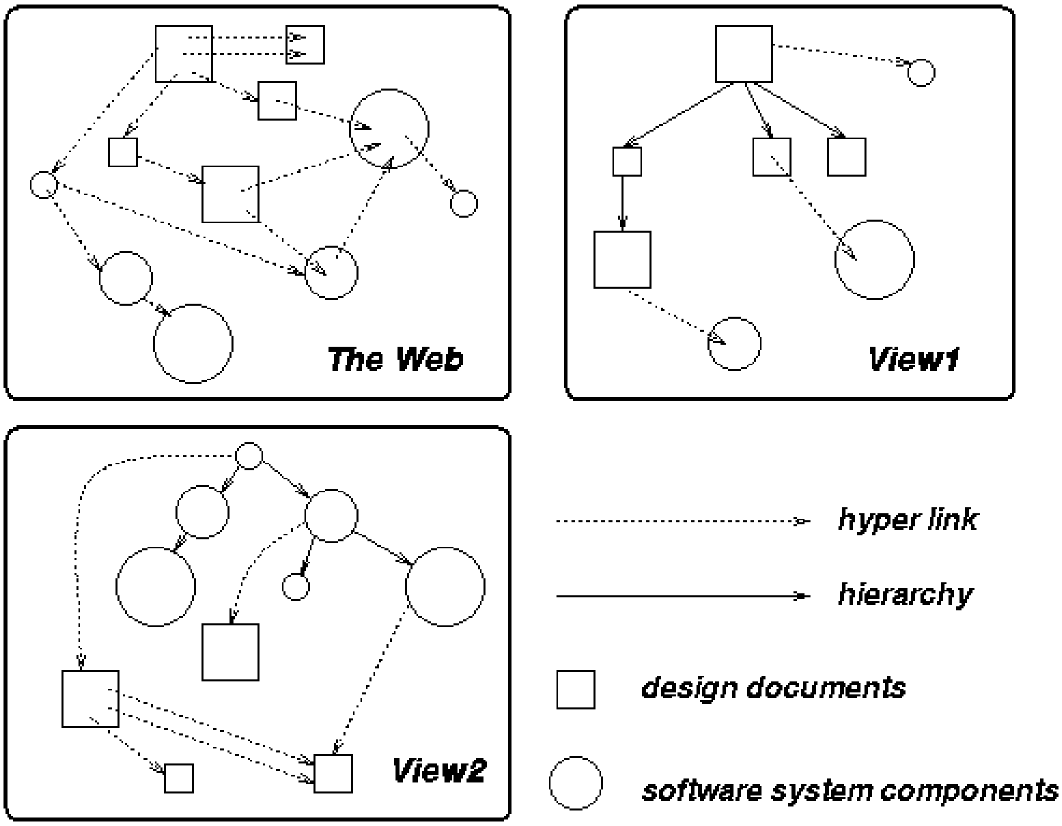

Figure 1

In a software development environment that uses WWW for storage of design documents and source code, the designers and programmers would like to view the software system under development in different ways. The designers should see the system primarily as chapters and sections of design documents, perhaps with hyperlinks to corresponding code. Programmers, on the other hand, should see the system in terms of subsystems, modules and source files, perhaps with hyperlinks to relevant fragments of the design. These two sets of objects may be logically related in hierarchical structures illustrated in Figure 1.

Finally, suppose the web is used in a training system where an experienced user sets up a "trail" of web visits and operations for novice users to exercise. The view then consists of a temporal series of visits to web pages and encapsulates the operations (input) of the "teacher". To minimize authoring effort, the view might be constructed dynamically by tracing the experienced user's web traversal.

Requirements

The main way to provide customized views of the web is currently through "homepages" with hyperlinks to web pages of interest to the homepage author. A homepage may include queries to search engines (Alta Vista, Lycos, Yahoo, etc.), with limited tracking of web changes, but is relatively static with the hardcoded URLs generally modified by hand. Homepages alone cannot provide application-specific (and thus more powerful) query ability.

We identify some essential requirements for dynamically customizable web views:

- ***Dynamic***
  It is a highly dynamic world: the information on the web often changes without notice. This includes modification of content, movement of web objects, emergence of new web objects, etc. Therefore, views must be able to encompass changing data with relatively low cost. Other than the generic information retrieval engines, most structuring systems provide only static views.

- ***Global***
  The system should not be restricted to a particular website, nor even to the web. Applications may need to integrate information from local objectbases or other repositories. The users should be able to define and customize web views for the two kinds of data in a consistent way.

- ***Powerful***
  The system needs to provide users with powerful capabilities concerning not only search on the content of the web pages (using information retrieval techniques and keywords) but also the attributes of and relationships among web pages (using database-style queries). For example, a user may need to find all web pages that reference a given URL whose annotations include at least one containing a particular term and that have changed in the past three days.

- ***Browser Independence***
  Some existing systems tackle these problems by developing special-purpose browsers. There are two difficulties with this approach: First, some commercial packages (e.g., Eudora) work together only with well-known browsers (e.g., Netscape Navigator), therefore a new browser would mean poor interoperability. Second, this implies application-specific modifications to HTTP and/or HTML, and duplicate efforts for browser and httpd builders.

The architecture presented in this paper uses Object Oriented Databases (OODB) to perform the core functionalities including view and meta-information storage and query processing. OODBs provide a suitable data model for WWW artifacts: web data is presented by attributed objects with referential and composite relationships, in addition to page content and embedded hyperlinks. Also, the rich set of associative and/or navigational functions provided by most OODBs enables powerful search capabilities. Our architecture does not require modifications to current Web protocols or browsers.

System Architecture

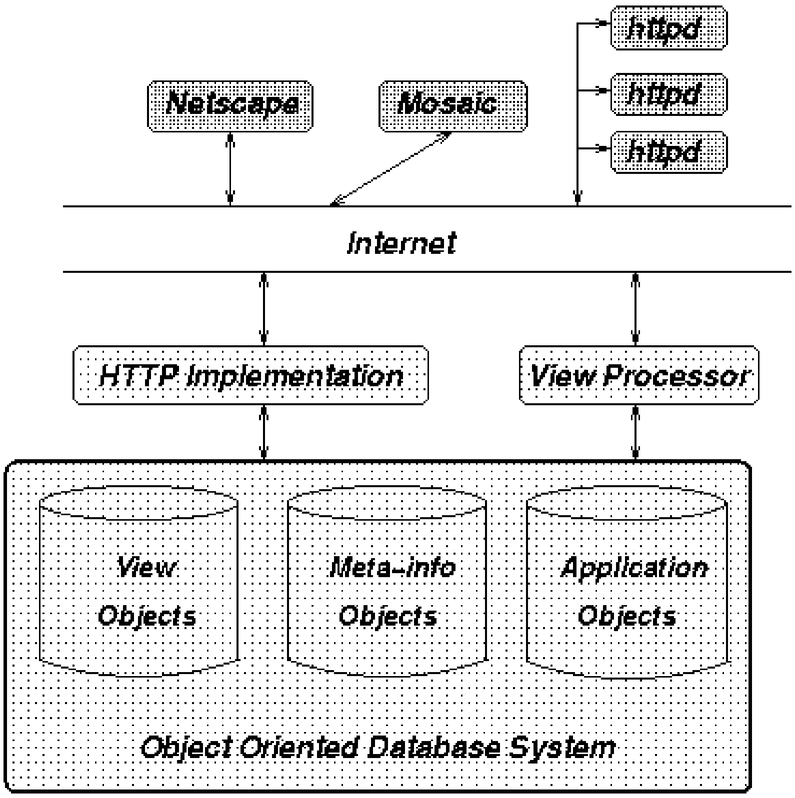

Figure 2

The system architecture is the key to supporting dynamically customizable views on the web. Our architecture is shown in Figure 2. It consists of three major components:

- ***HTTP Implementation***
  This component accepts requests from browsers through the HTTP protocol. It forwards the request to other parts of the system to obtain appropriate data and reply to the browsers. As we shall see, this component could be a standard HTTP server (httpd) in some implementations.

- ***Object-Oriented Database***
  Three different kinds of objects are stored in the database:

  - ***View Objects***
    These objects store view definitions. A view is defined in an enhanced HTML format and includes embedded objectbase queries that ultimately generate hyperlinks to web objects.

  - ***Meta-Information Objects***
    These objects model the part of the web of interest. Meta Information, which may include arbitrary information about the corresponding web object (or web objects if the mapping is not one to one), is represented as attributes of the meta-information object. They are queried by the View Objects to generate hyperlinks to web objects.

  - ***Application Objects***
    These objects are application-specific and not stored on the web.

- ***View Processor***
  The View Processor is the core processing engine of the system. It reads view definitions from the objectbase and perform the embedded queries to obtain the objects needed to construct a view. It contacts remote HTTP servers to retrieve web objects and queries the local objectbase for application objects. To support dynamically changing web data, the View Processor must be able to handle situations such as unavailable web objects due to network failures (e.g., by caching the most recently accessed version).

The system architecture is augmented by *embedded methods* for web objects. An embedded method is defined in the form of an "unknown tag" in HTML. The HTML specification defines that "unknown tags" should be ignored by browsers, therefore they will not be shown via a normal access. But when the web object is accessed by a View Processor, it may understand the information carried by the *embedded methods* -- which may be an attribute value that describes characteristics of the web object or an update operation on either the OODB or the web object itself (see below). Figure 3 shows a simple example.

```
<TITLE> Example of Embedded Methods </TITLE>
<H1> Example of Embedded Methods </H1>

<EMETH type=attribute name=author
         value=jyang@cs.columbia.edu>
<EMETH system=DKWEB type=operation name=NewComment
         npara=2 para1=from para2=content method=GET
         operator=http://www.cs.columbia.edu/~jyang/cgi-bin/sendmail.cgi>
...
```

Figure 3

Embedded methods is one way to turn web pages into *Web Objects*. Currently, most web pages are simply hyperlinked passive data. In order to treat the web as a huge OODB, we need to add on attribute values and operation methods as in other object systems.

In particular, *embedded methods* provide a way to define customized operations. In Figure 3, `NewComment` is implemented through sending email to the author; alternatively, the cgi-bin could do other things such as appending the comment to the page. The *embedded methods* should be written in some format expected by the View Processor. For example, the View Processor needs to identify the keywords `from` and `content` and give them appropriate meanings (in this case, the person who gives the comment and the comment itself). Different View Processors may interpret these methods differently (especially how to define parameters), which is a problem when the web page is used by arbitrary View Processors. Therefore, we propose to use the `system` keyword (such as in the NewComment example) to distinguish different protocols that may be understood by one or more Web View systems.

A Web View system goes through a number of main steps to construct a view:

Request Acceptance: The HTTP Implementation receives a request from a client browser in the form of a GET or POST method. GET and POST are the two major commands accepted by HTTP servers. In our approach, GET methods are usually used to obtain pre-defined views and POST methods to modify views. The HTTP implementation translates the requests into OODB queries or update commands, and forwards the queries or updates to the OODB or the View Processor depending on the type of the request.

View Retrieval/Update: The OODB gets requests from the HTTP Implementation and retrieves/updates view objects. If a request updates a view, the OODB informs the HTTP Implementation of the result and replies to the client browser, finishing the operation. Otherwise, the next two steps are invoked.

View Construction: After the view object is retrieved from the objectbase, it is sent to the View Processor, which parses the view definition and pulls in necessary data from either the local objectbase or the web. The View Processor queries the objectbase for the source location of the objects and either contacts local objectbases for application objects or the appropriate web site for remote web objects.

Reply: The View Processor then gives the processed view back to the HTTP Implementation, and the processed view is ultimately sent to the client browser.

The following is an example of how view objects and meta-infomation objects are defined:

```
# The class definition of View and MetaInfo
# They are inherited by specialized View and MetaInfo classes

View :: superclass ENTITY;
    owner      : User;
    public     : boolean;       # is this view public readable?
    content    : text;          # the view definition in enhanced HTML
end

MetaInfo :: superclass ENTITY;
    author      : string;       # the email of author
    URL         : string;
end

# example of subclasses of MetaInfo

DocRoot :: superclass MetaInfo;
    system      : string;       # name of the system that this document is f
    chapters    : set_of Chapter;
end

Chapter :: superclass MetaInfo;
    chapter_number : integer;
    module      : link Module;  # which module is related to it
    sections    : set_of Section;
end
```

In the objectbase, a view definition is stored for each view instance. The `content` attribute then contains the view definition written in enhanced HTML. The following is an example of view definition:

```
<TITLE> Example of View Definition </TITLE>
<H1> Example of View Definition </H1>

<HR>
This example view is a view to the software engineering environment.
The view contains the chapters and sections of the design document.

<P>
<DKOV_QUERY bind doc_root DOC_ROOT where (doc_root.system == "samplesys");>
<DKOV_QUERY bind chap CHAPTER where (member doc_root.chapters);>
```

```
The document of <DKOV_QUERY print doc_root> system contains the
following chapters:

<DKOV_QUERY print chap>
```

When the above view is queried, the View Processor reads the view definition and exeutes the queries as they are read, and plug in information when `print` commands are encountered. In this case the URLs that will get the doc_root object and chapter objects will be inserted.

However, the URLs point to objects (in this case MetaInfo objects) in the objectbase instead of their real URL. The reason is we need to insert other links into the web page before it is shown to the user. This is used to help the situations where there no hyperlinks in the original web object. For example, if there are no hyperlinks from each Chapter to the modules they describe, we can add them when the MetaInfo objects are referenced.

Variants

A highly centralized architecture would use an OODB to store virtually all the application-oriented information about the web objects in the corresponding meta-information objects in the database. Hierarchical and other relationships among meta-information objects reflect externally onto the underlying web objects. The web objects themselves are totally passive. They may provide some information to the OODB via *embedded methods*, but are not bound to, and in any case the OODB is not informed when they change. Therefore the OODB has to search for changes (including location movement) in the web crawling style of search engines. Cognizance of change is necessary when some meta-information is derived from the web objects' content or context (e.g., file system information like location, owner, permissions, creation time, last modification, etc.). Figure 4 illustrates this kind of system:

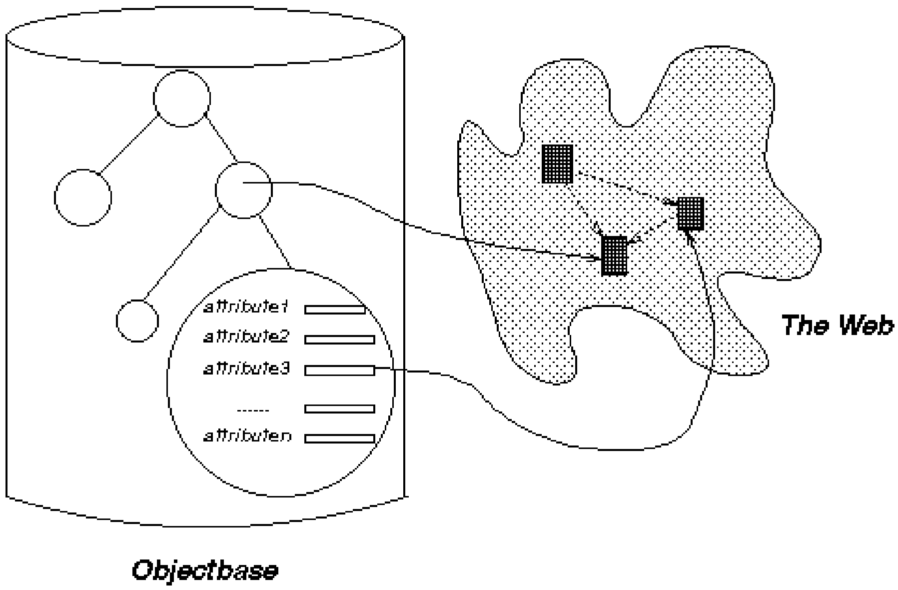

Figure 4

The main advantage of this model is simplicity. It requires minimum support from the web object owners: there need not be any *embedded methods* at all (although any that exist can still be exploited). The main disadvantage is lack of scalability. Depending on the application, the objectbase might manage a huge number of web objects, each with a substantial amount of meta information, and be responsible for detecting all their changes.

In a highly distributed architecture, the OODB degenerates to nothing and the View Processor might be just a slightly enhanced proxy server, as shown in Figure 5. All information about web objects is stored within their contents, and all relationships as internal hyperlinks.

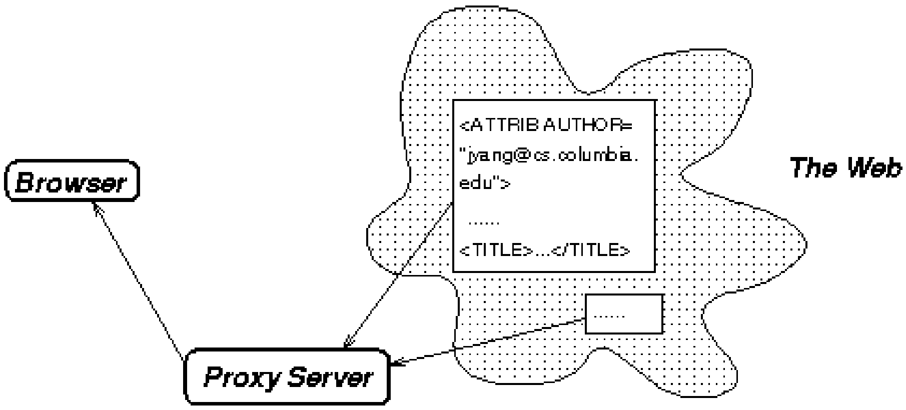

Figure 5

The main advantage of this model is scalability, since no information about web objects is duplicated. On the other hand, this approach depends on the web object owners to carefully program their web objects with embedded methods, if they are to achieve dynamically customizable views. This may be unrealistic, except when most web pages of interest are owned by technically-oriented persons - for example, in a software development environment where the users are designers, programmers and quality assurance personnel.

The third variant is a hybrid in between the other two. It uses OODBs to store meta information about web objects and also benefits from those web objects that are carefully programmed through *embedded methods*. The architecture is shown in Figure 6.

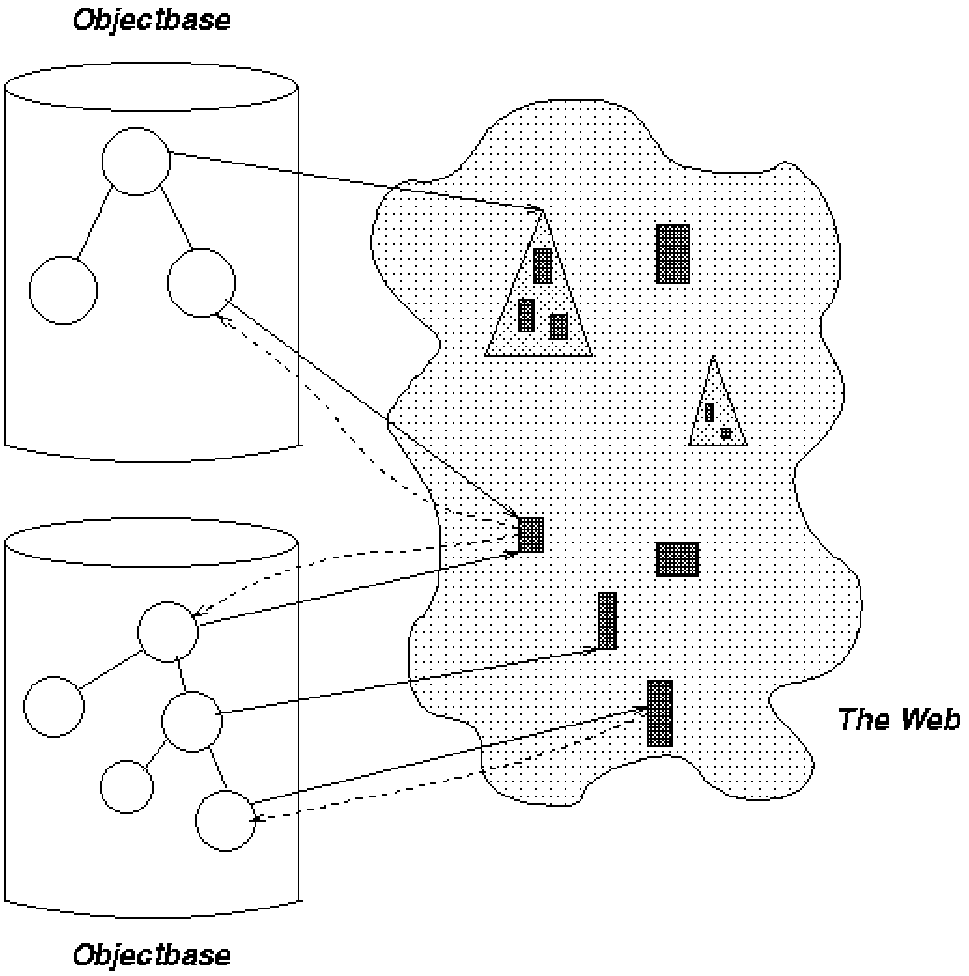

Figure 6

In this example, there are two independent OODBs operating on the same subset of the web. They store view objects and meta-information objects in order to provide rich facilities independent of the web objects. As in the first variant, an OODB object may point to a *set* of objects by maintaining a URL to a directory or a index page, increasing scalability (a one to one relationship between OODB objects and web pages is too costly when the number of web pages becomes very large).

But now, the web objects can be assumed to contain special links defined by embedded methods, which we call *backlinks*, to all those OODB objects that contain their meta-information. The *backlinks* could be implemented simply as URLs because the OODB objects are accessible via HTTP. *Backlinks* are useful for propagating web object changes among all OODBs that share it. When a web object is accessed through any one of the OODBs, its *backlinks* can be automatically followed so all OODBs that reference the web object can update their meta-information.

Implementation Issues

CGI vs. Server

In a CGI implementation, a standard HTTP server is a "front" for the OODB, with the CGI protocol used to submit objectbase queries. These queries may in turn generate further requests to other HTTP servers for web objects. The system is decomposed into several cgi-bins, each responsible for certain kinds of queries or updates. The advantage is the system is highly flexible in the sense that changing one or more of the cgi-bins easily changes the behaviour of the system. Also, commercial HTTP servers can be used so that their special features (e.g., secure HTTP) are still available.

In a server implementation, a standard HTTP server is not used for the HTTP Implementation, which is instead targeted to the specific application. This approach also has its advantages. Some possibly essential features of HTTP (such as the PUT method) are not implemented by most servers. It is generally more efficient, e.g., when a simple request is received, it need not spawn another operating system process or thread (to execute a CGI-bin) but just performs the request by itself. Perhaps even the OODB and/or View Processor are incorporated directly into the HTTP server. This may, of course, place an unacceptable load on the server process.

We also include the case of an HTTP proxy server. A proxy server is an intermediary designated by the client browser so that every HTTP request goes to it, and then the proxy fetches the URL from a conventional server and replies to the browser. A proxy server has the unique advantage of enabling tracking of clients who access web objects not mirrored in the objectbase. Because each access goes through the proxy server, the proxy can modify the web objects sent back to the browser. For example, one useful modification may be adding an extra form at the end of each web object to allow the user to add that web object into the objectbase on the fly.

Updating semi-local web objects

In applications where annotations or co-authoring is of interest, web objects may be the target of update actions from the View Processor. Although like other web objects, they are not stored in the local objectbase, per se, they may be considered part of the system (web objects are normally updated outside the system using various editing and file system facilities). To provide flexible, user-oriented schemas for updating what we call *semi-local* web objects, *embedded methods* are very useful.

For example, in a co-authoring system, each co-author might be allowed to overwrite the original web page. In this case, the web object would include an *embedded method*, recognized by the View Processor but not by a normal browser, to indicate the permitted update method. One possible *embedded method* would include the URL of a cgi-bin that simply replaces the page; a better method would first save the previous version using some version control tool like RCS or SCCS.

In an annotation system, constructing a comment might generate a new web object hyperlinked to the commented page. The latter might or might not be modified, by its *embedded method*, to include a hyperlink to the annotation; if not, the reference to the set of annotations could be represented instead in the OODB, perhaps with one meta-information object encapsulating the collection of annotations on the same original web object (perhaps covering also annotations on the annotations). In any case, the *embedded method* would need to specify where to store the new web object.

Using *embedded methods* to customize the operations allowed on a web object provides additional flexibility beyond that afforded by the PUT method (designed to write web objects), which is not left unimplemented by most HTTP servers.

Another advantage of *embedded methods* is that the owner of the web object can access the objectbase via the *embedded methods*. The simplest case is the *backlinks* discussed above. In general, the web object owner programs queries or update commands and the View Processor performs these objectbase operations whenever the web object is accessed. Note this provides a way for a web object owner to parameterize the operation of complex *embedded methods* according to the information currently stored in the objectbase -- probably the ultimate in dynamic customization.

Realization in dkweb

We have developed a sample implementation using a home-grown Object Oriented Database called *Darkover* (which was originally intended for an unrelated purpose). The sample system supports a shared web navigation and annotation system. It provides the following features:

- ***Dynamic and personal views of the web*** : Various users can have their own views onto the web. The system provides support for users to organize the web pages in the way they would like to see.

- ***Shared comments*** : Users can make annotations on the web objects they are interested in. The comments are stored and can be viewed by other users as well.

- ***Tracked Navigation*** : The system tracks all web accesses originated from a view and provides services such as adding a new web object to the objectbase through any browser.

- ***Editing*** : Pages on the web can be edited in two ways. If a web page provides an "update" method, the method is called. Otherwise, a modified copy is stored in the objectbase and can be retrieved later for further editing.

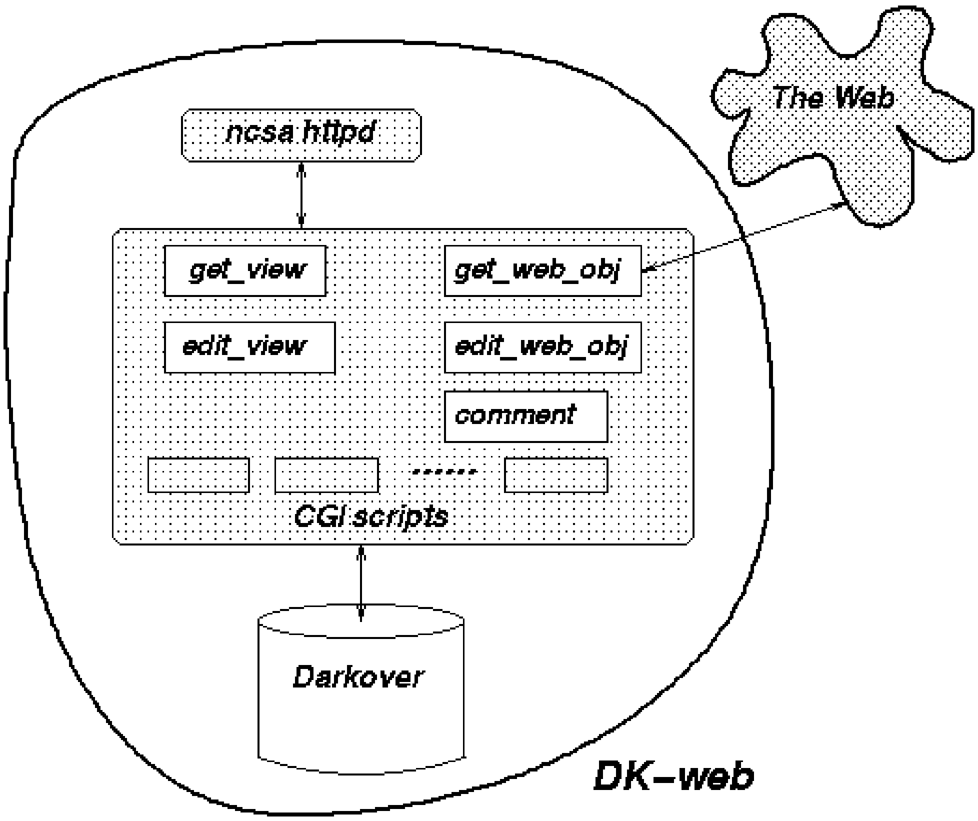

Figure 7

The implementation is illustrated in Figure 7. Several cgi-bin scripts perform the main functions of the system, and the *Darkover* objectbase supports a rich object-oriented data model and a sophisticated ad hoc query language (used in the scripts).

The `get_view` cgi-bin queries *Darkover* to find a view. In a view, each web object is shown as a hyperlink with some meta-information (e.g., last visited, size, etc.). When the user wants to retrieve the web object, he/she follows the hyperlink -- which is a reference to the `get_web_obj` cgi-bin. `get_web_obj` first looks at the objectbase to see if this is a local object, and fetches it from the home website if not. Then the script also parses the page and modifies all its hyperlinks so that further references are re-directed to this script. This ensures tracking of all browser accesses to web objects.

`get_web_obj` also adds three forms to the end of the page so that the user can:

- Add this page to the objectbase if it is not already there.
- Make comments on the page.
- Edit the source of the page. Unlike the comment form, the edit form isn't displayed unless the current user has update permission for the web object. Access permissions are modeled in the objectbase as attributes of the meta-information object that represents the web object.

The `edit_web_obj` cgi-bin is called when the edit form at the bottom of a web object is submitted. It checks whether the web object defines its own *embedded method* for update: If so, the edited source is provided to the method as one of its parameters. Otherwise, the modified version is stored in the objectbase. Later on, when other references to this web page are made, the modified version appear attached to the end of the original page via a hyperlink so that other users can see the modification as well (but the original page is **not** replaced).

Comments are stored in the objectbase as descendants of the composite meta-information objects mirroring the web object. But if the web object defines an *embedded method* named `NewComment`, all comments are forwarded to the owner (or otherwise handled) by that method.

Related Work

O2WebGateway and many other gateway systems support access *from* the Web *to* a database, but not vice versa. They demonstrate how to interface databases to the web but don't treat web pages as entities in an OODB or other kind of database.

Ockerbloom proposed the Typed Object Model alternative to MIME types whereby object types exported from anywhere on the Internet can be registered in "type oracles", specialized servers that may communicate among themselves to uncover the definitions of types registered elsewhere. Web clients who happen upon a type they do not understand can ask one of the type oracles how to convert it into a known supertype. Such typing facilities would nicely complement our representation of web objects in OODBs, and would enrich the modeling approach.

OreO is a proxy server-based architecture providing general-purpose information filtering between web browsers and HTTP servers. It presents a novel way to support static views and interoperability among the views. OreO's focus is on "constructing highly specialized transducers that can be composed to produce more sophisticated aggregate behavior". The aggregate behavior, however, does not address the dynamic property of the web views presented in this paper.

ComMentor supports sharing of in-place annotations attached to arbitrary web pages. The approach depends on specialized browsers so that a more complicated client/server communication protocol can be leveraged. We noted some disadvantages of customized browsers above. The ComMentor demo we observed depends on the remote display feature of X11 Windows (xhost). This has two disadvantages: first of all, PC users without an X-server simply cannot display the browser. Second, allowing remote access to users' displays introduces many possible security holes, e.g., one could use commands like xwd to dump a user's screen, write an X program to steal all keystrokes, etc.

Java supports the programming for the Internet in the form of platform-independent Java applets. It is quite different from what embedded method does. The embedded methods are used to supports web-object side operations. It is executed on the machine where the web object is from, while the Java applets are executed in the reader's machine. A roundabout approach to implement the Embedded Methods using Java is to have a front end Java applet that talks to a server on the machine where the web object is from. But since it is less efficient in execution and more time consuming for the web-object owners to write the applet *and* the server, it is not recommended unless sophisticated user interaction is needed for the method.

Conclusions

We described an architecture for integrating OODBs with WWW to support dynamically customizable views on the web. The architecture depends on several new ideas, most notably *embedded methods*. We discussed architectural alternatives and implementation issues, and briefly sketched our implementation of **dkweb** based on a conventional OODB. The OODB (i.e., Darkover) is not altered in any way to support **dkweb**, which means this approach could apply to an arbitrary OODB.

References

1. Programming Systems Lab, *Darkover 1.0 Manual*, Columbia University Department of Computer Science, CUCS-023-95e, March 1995, ftp://ftp.cs.columbia.edu/pub/marvel/oz.1.0.manuals/IV.Darkover_API/.
2. Israel Ben-Shaul and Gail E. Kaiser, *A Paradigm for Decentralized Process Modeling*, Kluwer Academic Publishers, Boston MA, 1995.
3. Tim Berners-Lee and Robert Cailliau, World Wide Web Proposal for a HyperText Project, *CERN European Laboratory for Particle Physics*, Geneva CH, November 1990, http://www.w3.org/hypertext/WWW/Proposal.html.
4. Stephen E. Dossick and Gail E. Kaiser, *WWW Access to Legacy Client/Server Applications*, January 1996, submitted to this conference, http://www.cs.columbia.edu/~sdossick/www5.html
5. Jean-Claude Mamou, *OBDC and Mosaic*, October 1995, http://www.w3.org/hypertext/WWW/Gateways/OQL.html.
6. John Ockerbloom, *Introducing Structured Data Types into Internet-scale Information Systems*, Carnegie Mellon University School of Computer Science, May 1994, PhD Thesis Proposal, http://www.cs.cmu.edu/afs/cs.cmu.edu/user/spok/www/proposal.html.
7. Charles Brooks, Murray S. Mazer, Scott Meeks, and Jim Miller, Application-Specific Proxy Servers as HTTP Stream Transducers, *4th International World Wide Web Conference*, Boston MA, December 1995, http://www.osf.org/www/waiba/papers/www4oreo.htm.
8. Martin Roscheisen, Christian Mogensen and Terry Winograd, *Scalable Architecture for Shared Web Annotations as a Platform for Value-Added Providers*, October 1995, http://www-pcd.stanford.edu/COMMENTOR/.
9. Ari Luotonen and Kevin Altis, World-Wide Web Proxies,*First International World Wide Web Conference*, Geneva CH, May 1994, http://www.w3.org/hypertext/WWW/Proxies/.
10. *The Common Gateway Interface,* http://hoohoo.ncsa.uiuc.edu/cgi/overview.html
11. James Gosling and Henry Mcgilton, *The Java(tm) Language Environment: A White Paper,* http://java.sun.com/whitePaper/java-whitepaper-1.html

About the Authors

Jack Jingshuang Yang
Ph.D. candidate
Columbia University Department of Computer Science
500 W. 120th St. Rm. 450
New York, NY 10027
jyang@cs.columbia.edu

Gail E. Kaiser
Associate Professor
Columbia University Department of Computer Science
500 W. 120th St. Rm. 450
New York, NY 10027

kaiser@cs.columbia.edu

Research Credits

The Programming Systems Laboratory is supported in part by Advanced Research Project Agency under ARPA Order B128 monitored by Air Force Rome Lab F30602-94-C-0197, in part by National Science Foundation CCR-9301092, and in part by New York State Science and Technology Foundation Center for Advanced Technology in High Performance Computing and Communications in Healthcare NYSSTF-CAT-95013.

The views and conclusions 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 US or NYS government, ARPA, Air Force, NSF, or NYSSTF.