JavaObjectWeb:

Web-Based Hypermedia Storage

for

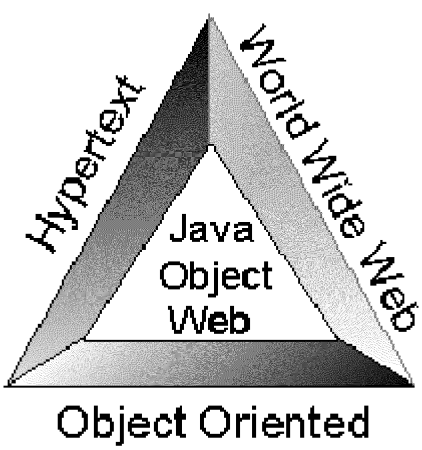
Dynamic Java Objects

John B. Smith, F. Donelson Smith,

Qian Li, Yufei Qian, and Yuqian Tu

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

jbs@cs.unc.edu
919-962-1792
919-962-1799 fax



0. Introduction

A. Overview

Our research is located at the intersection of three technologies:

1. Hypertext Systems
2. The World Wide Web
3. Object-Oriented Programming

We are developing a general object storage and access system with an architecture that incorporates many of the best features found in these technologies in order to create a flexible and scalable design. The result is a design that is both powerful and coherent, yet one that complements and is compatible with all three technologies.

Our system is called JavaObjectWeb to suggest several key ideas. This system creates multiple, interconnected regions of the WWW name space where additional functions and semantics exist. It is based on object-oriented principles, in general, and Java in particular. It provides facilities to support authoring and maintaining web content, including reliable and robust links. It also provides a framework in which arbitrary types of objects may be displayed and edited, including support for dynamic delivery of required Java classes to users.

To date, our research has produced the following results: (1) an architecture for a web-based storage, access, and retrieval system for Java objects that is based on hypertext graph semantics and can be used for a broad range of applications, (2) a proof-of-concept implementation, in Java, suitable for usability and performance evaluations of the system, and (3) an understanding of how design trade-offs effect performance and scale and of how to measure or estimate those effects.

The key questions we are addressing include: (1) What are the advantages and disadvantages of using linked Java objects for authoring and maintaining large-scale web content (as an alternative to today's use of typed files and markup languages)? (2) Can an implementation of the system's architecture scale and perform adequately for very large sites and large collections of related sites? (3) Can an implementation support dynamic reorganization of objects, maintain reliable and robust links among them, and still meet requirements for performance and scale?

The discussion here covers three aspects of our research. (1) the architecture of the system, (2) the proof-of-concept implementation of that architecture, and (3) our use of that implementation to explore issues of usability, compatibility, scale, and performance.

B. Motivation and Review of Prior Work

Our research is motivated by prior work in hypermedia systems, in general, the World Wide Web, in particular, and object-oriented programming. As indicated above, we are attempting to build on the strengths of these technologies while addressing several of their basic limitations. In this section. We briefly outline the evolution of several key concepts that we draw on and the issues or limitations we are attempting to address.

Hypermedia systems. Hypertext/hypermedia systems began with an idea described by Vannevar Bush in 1945, were first implemented by Doug Engelbart in the 1960's, were kept alive by researchers at Brown University during the 1970s, and flourished in the 1980s. The Augment system, developed by Engelbart and his colleagues in the 1960s [Engelbart, 1984], was the first implementation of Bush's [1945] original ideas for hypertext. During the 1970s, hypertext research was kept alive by Van Dam and his colleagues at Brown, culminating in Intemedia [Meyerowitz, 1986], a system notable for its emphasis on the visual interface and support for educational applications. During the mid-to-late 1980, hypertext systems flourished, in part because of the wide availability of personal computers. Some typical systems were Notecards [Halasz, 1988], a system for organizing ideas; Storyspace [Bolter, 1991], a system for creating fictional webs; KMS [Akscyn, *et.al.*, 1988], the first commercially viable hypertext system; WE [Smith, *et.al*, 1987], a system that incorporated a cognitive model of writing in its user interface and system designs; and ABC [Stotts, *et.al.*, 1994, Shackelford, *et.al.*, 1993, Smith, *et.al.*, 1991], a system supporting collaborative use of hypermedia materials. Common characteristics of all of these systems were their focus on and support for the process of organizing ideas and authoring hypertext documents and the fundamental requirement that links be maintained so that they could not be left dangling. They differed widely in the specific applications and populations of users for which they were intended. Whereas KMS and Intermedia were intended to be used by dozens to hundreds of users, the majority were intended for individuals or a small number of users.

Some key concepts found in many of these systems which are used in JavaObjectWeb include: (a) the organization of data as a set of nodes and relations among them as links, (b) visualization and direct manipulation of the resulting graph structure, (c) inherent support for authoring and maintaining content, including link reliability, and (d) access primarily through browsing and navigation rather than search or specification of file names. While many hypermedia systems offered interesting alternatives to file systems and databases, most were constrained in capacity and most were single platform systems.

The World Wide Web. Hypermedia during the 1990s has been dominated by the World Wide Web (herein referred to as *WWW* or the *Web*). The world of hypertext changed radically during the 1990s with the appearance of the World Wide Web [Berners-Lee, 1994], the first hypertext system conceived as an Internet application. Originally designed as a document delivery system for a few hundred scientists, the Web has become the user interface to a wide range of services and data types for hundreds of millions of people, world-wide. As a result, its scope is global, and the ramifications with respect to scale and numbers of users are well-known. Other ways in which the Web has changed the way we think about hypertext is its emphasis on cross-platform support through independent implementation of HTTP and HTML, and its incorporation of other Internet-related concepts (e.g., the MIME typing scheme) and services (e.g., FTP, e-mail, and news).

Features found in other hypertext systems that the Web omitted during its gestation period (1990-1993) may be as important in explaining its early success as those it included. Among those important omissions were reliability of links and explicit support for authoring and maintaining content. Whereas earlier hypermedia systems had assumed that it was essential to ensure the integrity of links when nodes are moved or rearranged, the Web simply ignored this issue. If a file was moved from one directory to another or renamed, links represented as HTML anchors with URLs would break, but so be it. Similarly, the Web largely ignored authoring support and left it to users to find other means to create and edit content files and insert them into a Web server's file space.

Since 1993, several hypertext systems and standards efforts have emerged that have attempted to address some of the perceived limitations of the original WWW architecture. The most notable of the alternative systems are Microcosm [Fountain, et.al., 1990] and Hyper-G [Maurer, 1996], since renamed "Hyperwave." Both of these system have included support for authoring as a basic requirement; both maintain links in a separate database in order to provide reliable links over restricted domains, and both support access through conventional WWW browsers. However, both offer only a small number of editing tools, developed as internal applications Neither architecture is easily extended, making the adding of new data types difficult. A important distinction between these systems (as well as the Web itself) and our architecture is that they view files as the fundamental unit of data whereas our architecture derives from O-O principles and supports objects as the fundamental entity.

A different emphasis can be seen in other efforts to address limitations in the Web and earlier hypertext systems through a new hypertext standard. That work began with the Dexter group [Halasz & Schwartz, 1994] and has evolved into an on-going effort to develop an open hypertext protocol. Several participants have tried to situate this perspective [Osterbye & Will, 1996] and to implement concepts from it [Will & Leggett], but the work of this group remains incomplete at this time.

We also note the World Wide Web Consortium's own efforts to address fundamental limitations in the Web' architecture. Some of the restrictions on authorship are starting to be addressed through HTML extensions (e.g., XML and DHTML), HTML editors, and server support for the HTTP POST method. Nevertheless, authoring new content is still not well integrated into the Web and few tools exist for visualizing, navigating, and directly manipulating the structure of large, complex, content collections and sites. They have also considered possible extensions to the architecture to provide reliable links [Ingram, et.al., 1996]; however, these considerations are as yet incomplete and implementation experience is limited.

In summary, as the WWW continues to grow in size and importance, more and more work is going into extending its functionality such that the Web has become the interface to the Internet. This generality requires that network-centric systems intended for general use, including hypertext systems, must remain compatible with the Web as it continues to evolve. Consequently, we regard Web compatibility of our system as a fundamental requirement.

Key concepts from the Web that our system is based upon are: (a) a universal addressing architecture for naming and locating information (URLs), (b) a design as a distributed, client/server system intended to be used by remote users through the Internet, (c) access from multiple platforms through independent implementations of a common protocol, (d) exploitation of the "point and click" metaphor within a GUI interface for access and navigation, and (e) support for multiple MIME types.

Object-Oriented programming. The third technology on which our research is based is object-oriented (O-O) programming. Some key concepts from the O-O perspective include encapsulating function and data within an object, reuse of components, and the incremental building of more specialized components from more general ones through inheritance. However, object-based systems have not provided many tools for browsing collections of objects or integrating persistent storage of objects into the Web. Our work extends basic O-O concepts by adding a hypertext-based object storage system implemented with Java [Gosling, et.al., 1996; Arnold & Gosling, 1996]. In the

Java environment in which we work, platform independence – the notion that the same program can run on many different hardware and software systems -- is also important.

Each of these technologies offers a number of useful features. Yet, each has also left out features found in the others that would make it even more useful if they could be added. How much more powerful would conventional hypertext systems have been if they had been integrated into the Internet and had achieved the scale shown to be possible by the Web? How much more useful would the Web be if all links remained valid so long as the target exists regardless of its location; if users could build new pages as easily as they can browse them; or if users could see and manipulate the logical structure of their sites through GUI direct manipulation tools? How much more useful would object-based systems be if persistent storage systems could be organized and accessed via a hypermedia system based on the Web?

We have addressed these limitations by creating a coherent, scalable architecture (that is fully compatible with the Web) for creating, storing, navigating, and maintaining content created as Java objects. Key concepts incorporated in the JavaObjectWeb architecture include the following:

1. Content is created, maintained, located, and viewed as (Java) objects.
2. Objects are typed (MIME).
3. Objects have identifiers that are globally unique, meaning that once created, they can be located regardless of where they are stored; thus, conventional HTML anchors/links based on URLs that include these identifiers never break.
4. Objects have their own editing and display methods that can be loaded dynamically from remote locations; consequently, all objects can be displayed and edited (and new ones created) even if object classes are not directly available locally.
5. Hypertext-based graphs constitute an important new class of object supported by the system. All content objects are organized into a collection of graph structures that may be displayed and restructured through direct GUI manipulation.
6. The object storage system is fully Web compatible and is implemented as an extension of conventional Web technologies.

1. System Description

In this section, we first describe the general architecture we have developed, then the data model which is the foundation for that architecture, and, finally, the design and implementation of our prototype system.

A. Architecture

The architecture is composed from three distinct functional groupings, each of which provides a useful subset of the total function. The conceptual layering of these three groupings is shown in Figure 1.

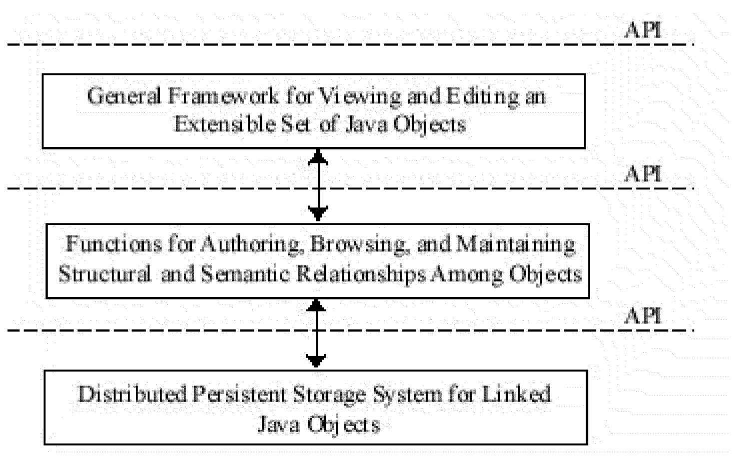


Figure 1. JavaObjectWeb's layered architecture.

The basic building block for this architecture, shown as the bottom layer, is the storage system for linked Java objects. Its data model and related functions are described in detail in the following section. The API supported by the storage system can be used in any application that requires persistent object storage. We have used it to create an application that supports authoring and reliable links within regions of the overall World Wide Web name space. We also use this API for the test tools described in Section 2, below. Other systems with requirements different from those of JavaObjectWeb could utilize the storage system as well.

An important idea we are exploring is that content can be comprised of objects, rather than relying solely on conventional files of MIME-typed data or markup languages. The middle layer of our system provides a general framework, based on Java object reflection, that allows arbitrary object types to be created along with viewers and editors for those types. A unique aspect of this design is that it uses the distributed storage system to locate and distribute, when and where needed, any Java classes required to display and edit an object at the time that object is accessed. This approach stands in contrast to the current practice of using preloaded plugins that are not part of the WWW architecture to enable a browser to display a given MIME type of data.

At the topmost layer of this architecture are functions that support the creation and maintenance of structural and semantic relationships among objects. It is based on storing, along with objects representing conventional data types (e.g., text or graphics), a set of graph objects in which the edges of the graph represent relationships among the nodes of the graph were each node represents a content object held in the storage system. Using the framework of the middle architectural layer, facilities are provided that give users (programs as well as people) the ability to create, browse, traverse (search), edit, and maintain the graph objects that define the overall structure of the collection.

B. Data Storage Model

Two fundamental principles underlie the data storage model: (1) the smallest addressable unit in the store is a *content object* of arbitrary (but specified) type, and (2) every content object in the storage system has a globally unique address which is also a valid URL and is, thus, completely compatible with the addressing scheme used in the World Wide Web.

The specific form of URLs used in the storage model includes conventional host[:port] components. However, instead of a path component, our system uses a 64-bit object identifier (OID) that is unique within a given region that is identified by the host[:port] components of the URL (regions are collections of administratively-related storage servers and are explained further in a later section). In the example below, the 64-bit OID is represented by 16 hexadecimal digits. (Note: human beings are not expected to read or create OIDs; they are maintained by the system software.) For example, the URL **http://jow.cs.unc.edu:8888/00D00A7001FE00C6** represents an object that is uniquely addressed by OID **00D00A7001FE00C6** within the region managed by a server process that implements the HTTP protocol and is running at port **8888** on host **jow.cs.unc.edu**. Once a content object is created, its URL is never changed and the included OID value is never reused even if the object is deleted. Although an OID may have internal structure that is used by the storage system for efficient access, it is treated as an "opaque" (uninterpreted) value by applications.

Content objects are composed of two parts: (1) a set of properties that uniquely define the characteristics of a specific type of object, and (2) data that represent the true "content" of the object. For example, an object's data content could be HTML text, GIF or JPEG images, digitized audio or video (MPEG), or any other form that can be created as a Java object. Among the important properties stored for a content object are its type (based on the MIME typing architecture [Borenstein & Freed, 1993]), specifications required for proper editing or viewing of the data content, and system-defined properties (such as creation time or ownership) that are maintained automatically by the storage system.

The data model provides three basic constructs for organizing information: graphs, the "contained" relationship between a node and a content object, and hypergraphs. The discussion that follows explains their use and our reasons for including them in the data model.

Graphs. All content objects are explicitly organized into graphs (stored as content objects of type:graph). This is done to provide a well-defined model for access and traversal of the object store. Graphs represent both structural and semantic relationships between nodes. This graph-based storage model explicitly encourages users to organize information according to principles of modularity and decomposition by making it easy to represent relationships among elemental objects. Research in hypertext/hypermedia systems has shown that this method of organizing information improves human comprehension and increases the potential for concurrent access to individual components. For example, the structuralrelationships among content objects that comprise a document might be represented by links that show the author's intended access ordering among text- or image-type objects that define sub-sections, sections, figures, and chapters. Semantic relationships might be represented as links (expressed either as HTML anchors or links in a hypergraph as explained below) connecting, for example, a text object describing a software concept to a figure object showing its design, to a class object giving its implementation, and to an image object depicting its user interface.

Because a link in a graph typically represents a structural relationship between two nodes in a collection of related nodes, we use the terms structural-link (abbreviated S-link) and structural graph (S-graph) to denote these concepts. (The reasons for this further delineation of graphs will become apparent when we discuss hypergraphs, below.) Nodes may have arbitrary numbers of in-coming and out-going S-links, including none. Thus, a common case is an S-graph containing nodes but no links, representing a set of related nodes that have no explicit structural relationships among themselves. S-links have a direction, although traversal is supported in either direction. The data model also provides a predefined set of strongly typed S-graphs. Currently five types are defined: general directed graphs, connected graphs, acyclic connected graphs, trees, and lists. Typed S-graphs are useful for dealing with issues such as integrity, consistency, and completeness in supporting tools for authoring and maintaining complex information structures. However, JavaObjectWeb relies on a client-side program -- i.e., the browser applet -- to support graph type constraints. Thus, from the perspective of the storage system, graphs, like other types of content objects, are opaque -- entities just to be stored and managed.

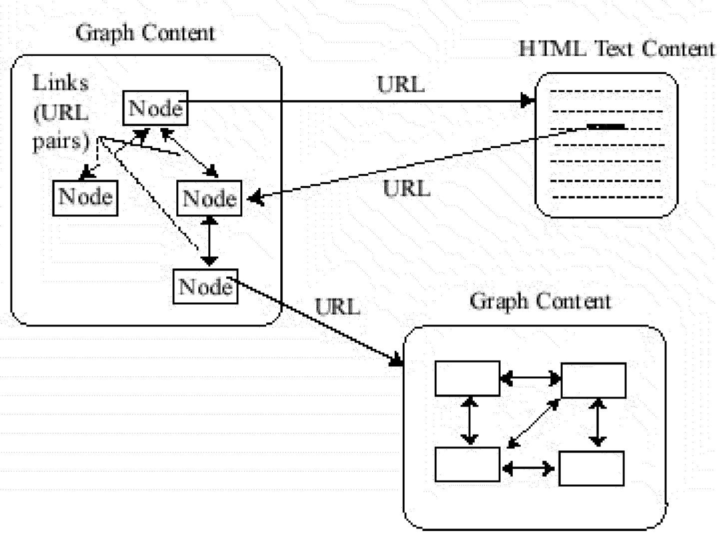


Figure 2. JavaObjectWeb supports two types of links. *Structural* links denote relationships between nodes in the same graph and are constrained by graph type semantics (e.g., tree). *Hypertextual* links denote relationships between nodes in different graphs and from anchors within HTML data; they are represented as special forms of URLs.

Node/Content relationship. Each node of the graph may contain the URL for an associated content object. We speak of this relationship as a "node containing another object" or as a "node and its content." We emphasize that this relationship between a node and a content object is an attribute/value relationship, as opposed to a link relationship between two nodes in a graph. Thus, whereas a node must exist in one and only one graph, a given content object may be referenced by an arbitrary number of nodes. Since a content object may be another graph, the data model is recursive, allowing hierarchical composition of a complex information structure from content objects. These relationships are illustrated in Figure 2.

Note that the data storage model subsumes the organization of data in a conventional file system: S-graphs are analogous to directories; and nodes, along with their respective data content objects, are analogous to files. However, the JavaObjectWeb data model provides additional functions for representing structure among nodes within an S-graph (directory).

While composition and structure are necessary for organizing complex information, they are not sufficient -- many useful relationships cannot be modeled as structure. The most obvious example is the fundamental role of anchor references in HTML files that create the cross-structure relationships which form the World Wide Web. Because all content objects in our system have valid URLs, these URLs may be used for anchor references in HTML files or in any other Web-based context. The practice of embedding URLs as anchor references in HTML files has, however, created serious problems in maintaining information in the Web (we note that there are other solutions to this problem, e.g., XML, but believe that the JavaObjectWeb data model represents an alternative solution with desirable properties). Unlike conventional Web servers which often invalidate embedded URLs when files are moved or deleted, the storage system described here provides stronger semantics for references to URLs. Specifically, it provides that (a) URL references to nodes that have been moved are always valid (this is accomplished with a forwarding mechanism), and (b) URL references to deletednodes return useful information, including the option for recovery of the related content object

Hypergraphs. To express semantic relationships within the data model analogous to HTML anchors with URLs, we also define a more flexible kind of link, called a hyperlink (H-link). Hyperlinks can represent any semantic relationship between two nodes. H-links are used for associations between nodes in different S-graphs or non-structural relationships between nodes within the same S-graph. This latter use permits links that would violate the type constraints of the particular graph type (e.g., tree), were they defined as S-links (see Figure 3). H-links and the nodes they link are grouped into hypergraphs (H-graphs). The properties of URLs, nodes, and links discussed above for structural graphs apply to hypergraphs, as well. Links similar in function to H-links are usually the key elements of conventional hypertext systems.

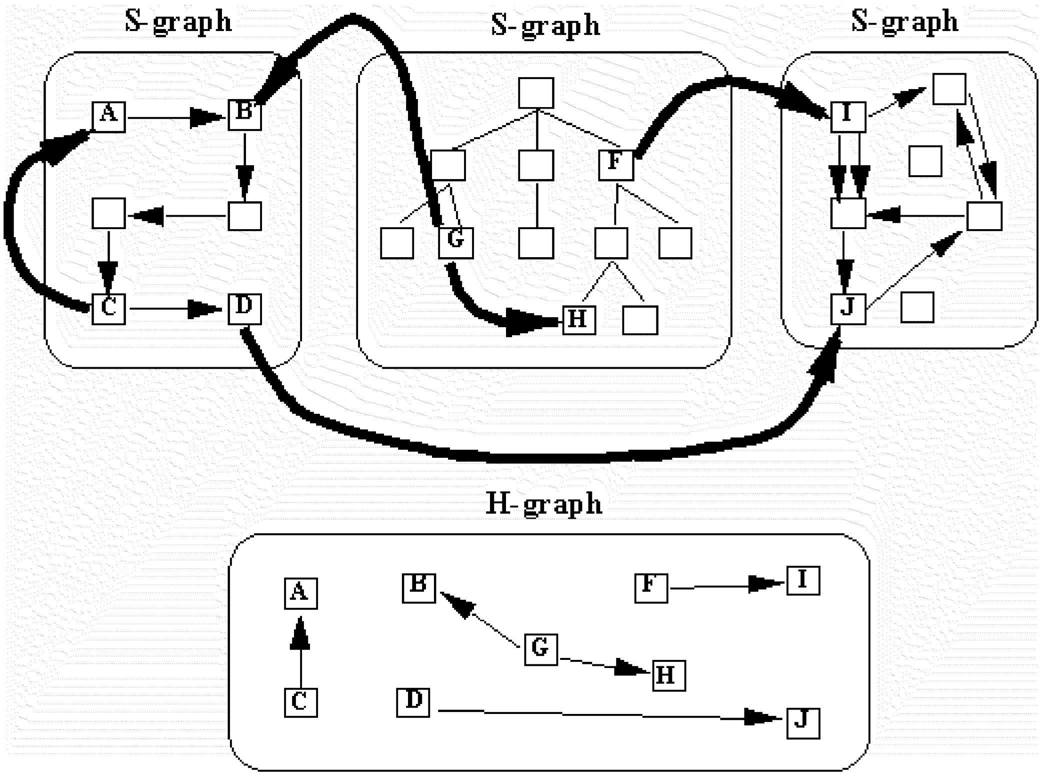


Figure 3. JavaObjectWeb's data storage model, showing both structural and hyperlinks. Sets of hyperlinks form *hypergraphs* .

C. Implementation

The goals for our implementation efforts were twofold. First, to create a proof-of-concept system showing that a Web-based object storage (built in Java) could provide the functions described above. Second, to show that the system could be used as a testbed to explore key issues, including scale, performance, extensible data types, and robust link maintenance even across multiple regions. We have completed an initial prototype and are now designing a second implementation. This section describes our original prototype. The main components of the system are shown in Figure 4.

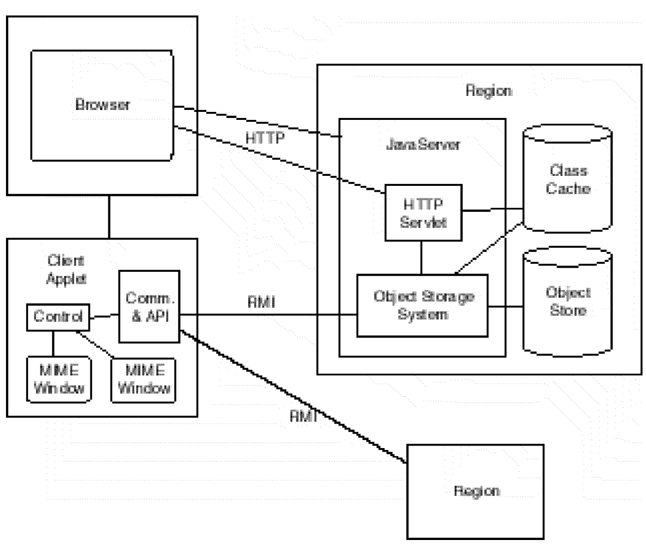


Figure 4. JavaObjectWeb's component architecture. It shows a client applet, loaded by a

conventional Web browser, communicating with a JavaObjectWeb server, running within the context of a JavaServer and comprising a *region*. Also shown is communication with multiple JavaObjectWeb regions.

The system can be divided into a client side and a server side. Both client and server include conventional WWW components to insure Web compatibility.

The JavaObjectWeb client The client side includes a conventional Web browser and a client applet that provides the user interface to the JavaObjectWeb system. The Web browser provides two functions. First, it allows a user to bootstrap the system. By selecting a well-known URL or a URL for any object within a JavaObjectWeb region, one of our servers will return an applet tag within conventional HTML that launches the client applet, providing a user interface. The second function is to render HTML data returned from the storage system; however, Java classes are now available for rendering HTML and we could eliminate this browser function to make the system more self-contained.

The client applet is responsible for launching new windows for specific data types, for enforcing constraints for specific graph types, for supporting cut/copy/paste operations between graphs, and for other point-of-control functions. During a typical session, a user will open and close a number of different windows to browse the structure of a region and to access particular content objects. This is done through direct connections with a JavaObjectWeb server using RMI (Remote Method Invocation), rather than through HTTP.

As described in the data model section, above, graphs play a particularly important role in our system. Consequently, visual renderings of graphs provide the primary metaphor for visualizing and manipulating logical structure. The windows for graph objects allow users to create new nodes through simple point and click operations, and users can reorganize the links that denote logical and semantic relations among nodes through simple drag and drop operations. Figure 5 shows a graph being displayed and edited within a JavaObjectWeb graph applet.

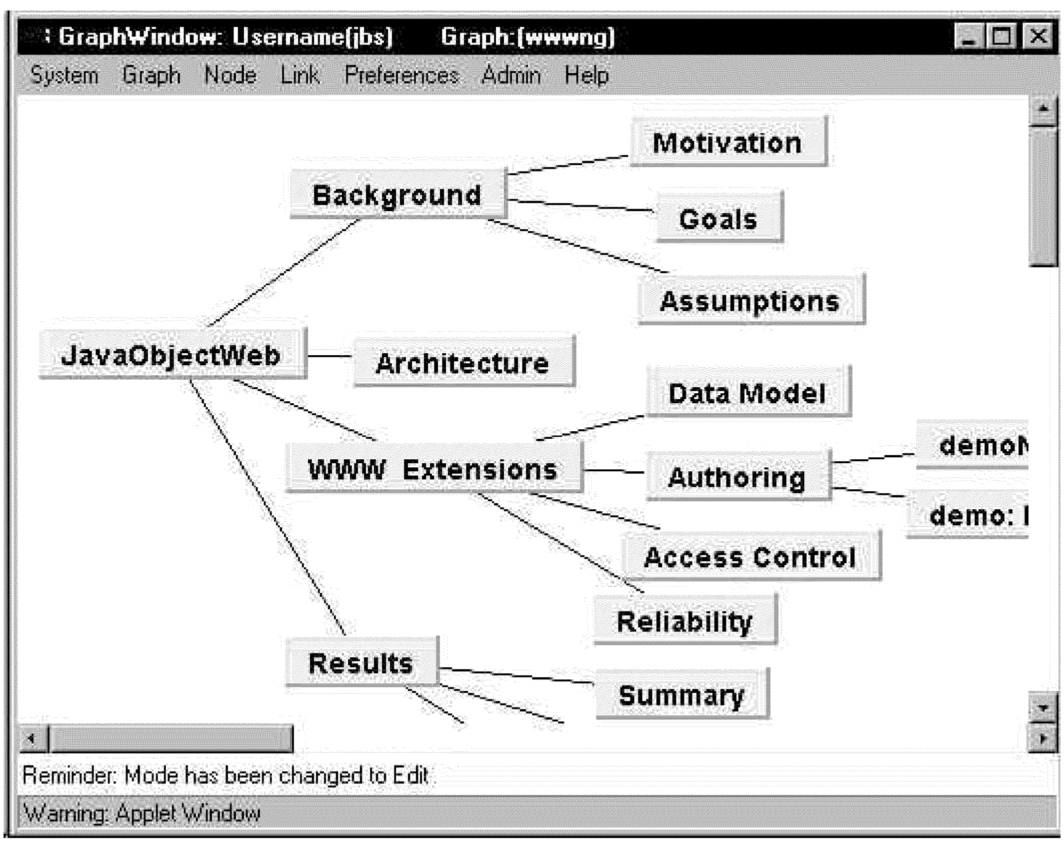


Figure 5. JavaObjectWeb graph applet. Shows a graph being displayed and edited. Nodes contain HTML text, JPEG images, and other graphs.

Data from non-graph content objects are displayed and edited in windows that implement the appropriate semantics for the data type. We currently support only HTML and JPEG as non-graph types. However, we are extending the implementation in the next version to support an open-ended set of data types. This can be accomplished by having the client query the content object for a URL that identifies the display and editing classes for that object type. It can then request a nearby server to fetch and cache them so they will be available for dynamic loading through the user's Java Virtual Machine (JVM).

The JavaObjectWeb server. The server side of our prototype system includes a JavaServer and several servlets that implement the storage system. Multiple instances of these components (typically running on multiple server machines) constitute what we refer to as a JavaObjectWebregion.

The JavaServer provides two main functions. Like the role of the Web Browser on the client side, the JavaServer insures WWW compatibility, and it is used to bootstrap the client applet. The second major function is to support servlets, which are used to implement the object storage system.

The object storage system provides several services. Its most important function is providing persistent storage of Java content objects. Those objects can be accessed through conventional HTTP requests, which return conventional HTTP messages with the content object's data included in the body of the message. More often, however, content objects are accessed through direct RMI connections from client applets. The system also provides strong access control and concurrency control, and it locates and caches the Java display and editing classes needed to support arbitrary object types in the client applet.

In order to provide scalability that could, potentially, approximate that of the World Wide Web, JavaObjectWeb's architecture for a region is more complex than that shown in Figure 4, above. In fact, JOW regions can contain multiple servers, as indicated in Figure 6. Such a region might correspond to a site or large department within an organization. The organization as a whole might be supported by a number of such regions world-wide. The JavaObjectWeb namespace is comprised of an arbitrary number of such regions, so long as they are accessible through the Internet.

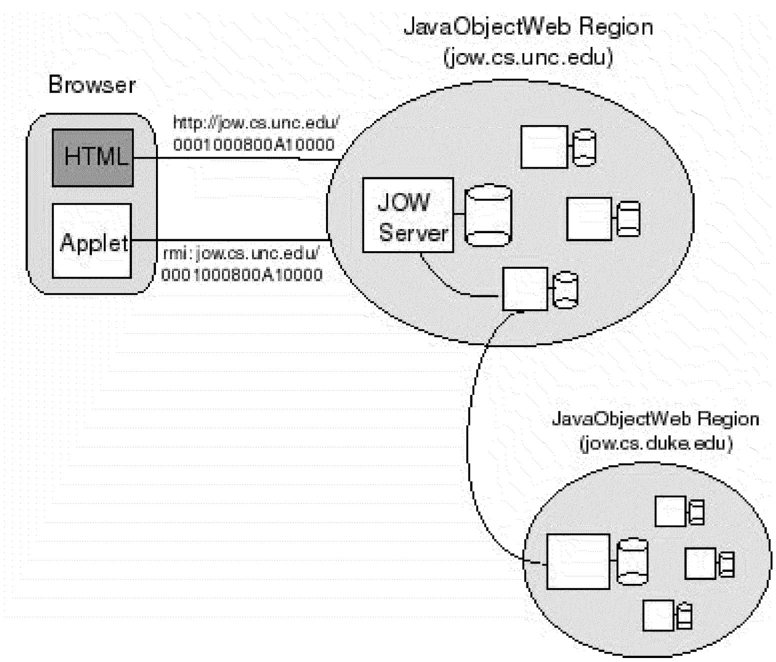


Figure 6. JavaObjectWeb's multi-region architecture. It shows a client applet communicating with the primary JavaObjectWeb server for a region. That server has forwarded a request to another server in the region which, in turn, has forwarded it to a server in another region.

Each region includes a primary server that provides the default access point to that region's data. Additional servers may be added at will. Servers can be configured to support the data for particular individuals or groups. However, content objects residing on any of the servers may be linked into graph structures, regardless of where those graphs are stored. Links and URLs to a node or content object "follow" it when it is moved from one graph to another, from one server in a region to another server, or from one region to another. Links in the form of HTML anchors using JavaObjectWeb URLs and system-defined hyperlinks will remain valid even when objects are moved from one location to another (logically or physically).

Objects are accessible through any particular server the user happens to be connected to, regardless of where the object is actually stored. Both general access and maintenance of link integrity for objects that have been moved are provided by functions built into each server that allow it to return objects stored locally but forward requests to other servers when they are not. When a distant server responds to a forwarded request, it returns the object to the original server which then returns it to the user.

2. Evaluation

An important dimension of our work is the claim that the architecture is scalable and can provide performance comparable to current WWW servers while providing additional function. Consequently, we are conducting several empirical investigations to evaluate these hypotheses. We discuss, below, our results to date. Although they are incomplete, we believe they offer several interesting insights.

By scalability we mean that the number of users that can be supported by fixed resources (e.g., a single server machine) has a wide range and that additional resources (servers) can be added incrementally to support large numbers of users. One critical aspect of scalability concerns the efficiency and performance of the implementation for links that remain valid when objects are moved or reorganized in the storage system.

Generating large webs of objects. To support our experiments, we needed a large collection of stored objects including the graphs that reflect structural and semantic relationships among objects. We developed an import program that processes the file system used by a conventional Web server to generate as output a corresponding organization expressed as JavaObjectWeb objects and graphs. Thus, we transformed a conventional collection of HTML, GIF and JPEG files into a JavaObjectWeb hypertextual structure of Java objects.

Generating access requests. To evaluate scalability we must be able to generate request loads on a server (or set of servers) that resemble loads that would result from large numbers of users of the system. To do this, we used benchmark programs running on multiple client machines, each instance generating requests by sampling at random from distribution functions that characterize the behavior of a user population (note that this technique has been widely used to evaluate the performance and scalability of distributed file systems). Each client machine runs multiple threads (up to 256), each simulating a different user's requests. These benchmark programs run under Windows NT on Intel architecture machines of 200 MHz or faster.

Experiments. With these tools for generating large object stores and for generating user requests, we have begun a series of experimental runs to test scale and performance.

Some of the questions we are attempting to answer include the following:

1. How many "typical" users can a single server support?

2. How does the performance of the system change as a function of the number, sizes, and types of stored objects and of the complexity of the structural and semantic links?

3. How does performance scale as more server instances are added to support a logical region?

4. How does the frequency of object movement or reorganization (e.g., the ratio of forwarded links) affect

performance?

To date, we have run only those tests involving a single server. Each client client machine generates requests based on an average "think time," exponentially distributed with means of 10 seconds for a graph and 20 seconds for an HTML content object. In the next phases of our work, we will extend the runs to include multiple servers (3 - 4) in the same JavaObjectWeb region, typically a single location (e.g., several departments at a single university). After that, we will test the system over several (3 - 4) such regions, such as departmental systems at several universities located across the country.

Results.

Although we are still in the early stages of our experimental tests, they have been very useful to us as we attempt to differentiate between issues of design that are inherent in our architecture and issues of implementation that are specific to our current prototype system, to Java, and to our particular computing and communication environment. Our early tests were intended to answer the first two questions, listed above.

We found that our current JavaObjectWeb server (a Sun Ultra Enterprise 3000 four processor machine with 384 MB of memory running Solaris) performed satisfactorily with respect to response times for approximately 100 users, but performance degraded quickly beyond that. These data, shown in Table 1, sent us back to the code to explain these unexpected results. We found that it is critical where Java object serialization occurs (i.e., client vs. server), particularly for complex objects comprised of many smaller objects. Our current implementation deserializes graph objects in the server, creating a bottleneck; we will move this function to the client to take advantage of "free" cycles on users' workstations.

Objects/Users	32	64	128	256
200 - 300KB	3.003	3.14	5.101	8.22
100 - 200KB	1.474	1.88	2.465	5.198
50 - 100KB	0.91	1.096	1.495	3.638
10 - 50KB	0.599	0.602	0.832	2.593
5 - 10KB	0.215	0.248	0.261	2.013
1 - 5KB	0.263	0.157	0.201	1.897
graph	0.409	1.464	5.815	46.389

Table 1. Access times in seconds for 32 to 256 simulated users, running as separate threads on 8 machines. Non-graph content objects are categorized by size, in ranges from 1 - 5KB to 200 - 300KB. Graph content objects are listed separately.

In a second series of tests, we factored out graph object serialization by referencing only non-graph content objects. These data, shown in Table 2, indicated that the server could provide reasonable response to some 300 or so users. Again, we were disappointed. A further look at the implementation revealed our over-zealous use of Java's synchronization feature with respect to object access. In effect, our current implementation allows only a single request to access a stored object at any given time, even for read access. When we eliminate this bottle neck, we expect performance to increase substantially, perhaps as much as a factor of ten.

Objects/Users	32	64	128	256	512
200 - 300KB	2.26	2.694	3.513	6.034	37.592
100 - 200KB	1.297	1.786	2.422	4.117	35.333
50 - 100KB	1.209	1.257	1.372	2.97	34.195
10 - 50KB	0.577	0.53	0.831	2.172	33.254
5 - 10KB	0.219	0.296	0.301	1.553	32.648
1 - 5KB	0.135	0.148	0.254	1.489	32.554

Table 2. Access times for 32 to 512 simulated users, running as separate threads on 8 machines. Requests include only non-graph content objects. Results are categorized by size, running from 1 - 5KB to 200 - 300KB.

Answers to questions 3 and 4, regarding multiple servers in the same region and multiple regions, respectively, must await further tests. The current implementation supports multiple servers within a region as well as multiple regions. However, we have not yet developed a sufficiently large and interconnected distributed test data store to provide meaningful results. We also need to complete the modifications to our current implementation, as outlined above. When these steps are completed, we will run tests on both distributed configurations. (We expect to finish our experiments prior to WWW8.)

3. Future Research.

The current status of our work is this. We have developed an architecture based on the ideas presented here. We have implemented most of our initial design in the form of a proof-of-concept prototype. We have completed an initial set of tests on a single JavaObjectWeb server. We have not yet implemented the dynamic class-caching scheme.

In future work, we will complete the initial planned set of tests to evaluate the architecture over multiple server regions and multiple region data stores. We will complete the implementation of the dynamic class loading feature that will enable JavaObjectWeb to support arbitrary types of data as content objects. We will also make the system more robust so that we may move from benchmark experiments, such as those described above, to user evaluations. Following those results, we plan to update the design and to build a second implementation to support actual-use evaluation over extended periods of time for significant numbers of distributed users.

While not the focus of this discussion, the storage system can support asynchronous collaboration through its strong access control and concurrency control components. In future work we expect to add support for collaboration and cooperative work through the browsing, direct manipulation, and authoring components. A key set of issues will be to monitor and evaluate the capabilities of the system to support collaborating groups working together from remote locations. This pattern of use will stress JavaObjectWeb’s link forwarding and maintenance components. As part of that line of work, we will also explore strategies for replicating parts of the data store and maintaining consistency among the replicas.

4. References Cited

Arnold, K.; & Gosling, J., (1996), The Java programming language. Addison-Wesley, Reading, MA.

Akscyn, R.M.; McCracken, D.L.; & Yoder, E.A. (1988), KMS: A distributed hypermedia system for managing knowledge in organizations. CACM, 31, 7, 820-835.

Berners-Lee, T.; Cailliau, R.; Lultonen, A.; Nielsen, H.F.; & Secret, A. (1994), The World-Wide Web. CACM, 37, 8, 76-82.

Bolter, J.D., (1991), Writing Space: The computer, hypertext, and the history of writing. Lawrence Erlbaum and Associates, Hillsdale, NJ.

Borenstein, N.; & Freed, N., (1993), Mechanisms for specifying and describing the format of Internet message bodies. IETF Network Working Group RFC 1521, http:// www.roxen.com/rfc/rfc1521.html.

Engelbart, D.C., (1984), Authorship provisions in AUGMENT. Proceedings of COMPCON'84, pp. 465-472.

Fountain, A.; Hall, W.; Heath, I.; & Davis, H.C. (1990). Microcosm: An open model with dynamic linking. Proceedings of European Conference on Hypertext, pp. 298-311.

Gosling, J; Joy, B.; & Steele, G., (1996), The Java language specification. Addison-Wesley, Reading, MA.

Halasz, F.G.; & Schwartz, M. (1994), The Dexter hypertext reference model. CACM, 37, 2, 30-39.

Halasz, F.G. (1988), Reflections on Notecards: Seven issues for the next generation of hypermedia systems. CACM, 31,7, 836-852.

Ingram, D.; Caughey, S.; & Little, M. (1996), Fixing the "broken-link" problem: The W3Objects approach. Proceedings for the Fifth International WWW Conference, pp. 1255-1268.

Maurer, H., (1996), Hyper-G: The next generation Web solution. Addison-Wesley, Reading, MA.

Meyerowitz, N., (1986), Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework. Proceedings of OOPSLA'86, pp. 186-201.

Microcosm (1998), URL: http://www.multicosm.com/.

Hyperwave (1998), URL: http://www.hyperwave.de/.

Osterbye, K.; & Will, U.F., (1996), The flag taxonomy of open hypermedia systems. Proceedings of Hypertext'96, pp. 129-139.

Shackelford, D.E., Smith, J.B.; & Smith, F.D. (1993), "The Architecture and Implementation of a Distributed Hypermedia Storage System," Proceedings of Hypertext '93, New York: ACM Press, 1–13.

Smith, J.B.; & Smith, F.D., (1991), ABC: a hypertext system for artifact-based collaboration. Proceedings of Hypertext'91, pp. 179-192.

Smith, J.B.; Weiss, S.F.; Ferguson, G.J.; Bolter, J.D.; Lansman, M.; & Beard, D.B., (1987), WE: A writing environment for professionsals. Proceedings of the National Computer Conference, pp. 725-736.

Stotts, D.; Smith, J.B.; Dewan, P.; Jeffay, K.; Smith, F.D.; Smith, D.; Weiss, S.; Coggins, J.; & Oliver, W., (1994), A Patterned Injury Digital Library for Collaborative Forensic Medicine, Proceedings of Digital Libraries '94, The First Annual Conference on the Theory and Practice of Digital Libraries, College Station, TX, 25-33.

Will, U.K.; & Leggett, J.J., (1996), The HyperDisco approach to open hypermedia systems. Proceedings of Hypertext'96, 140-148.