




Integrating Diverse Information Repositories: A Distributed Hypertext Approach

John Noll and Walt Scacchi, University of Southern California



**A distributed
hypertext architecture
provides transparent
access to autonomous,
heterogeneous
information
repositories. The result
is a powerful
organizational tool and
a simple yet effective
integration mechanism.**

In today's networked computing environment, powerful workstations support a variety of sophisticated, graphics-oriented applications. Local area networks connect these workstations to file servers; the LANs are themselves linked by wide area networks, enabling access to information around the globe. This abundance leads to diverse access protocols, storage managers, data formats, and user interfaces — rendering much data inaccessible to any single user simply because too many things must be known even to begin. Furthermore, finding an item once is not necessarily the same as finding it again. The original discovery process may be lengthy and involved, or even accidental.

How can we provide transparent access to heterogeneous information repositories, while maintaining their autonomy? In this article, we address key problems of support for multiple, heterogeneous repositories, each under separate and autonomous administration with a variety of incompatible interfaces; diverse, unconventional data types; and different ways of viewing relations among the same information items. We present a solution to these problems that is radically different from existing systems. It is based on our distributed hypertext (DHT) architecture, which combines transparent access to autonomous, heterogeneous information repositories and a powerful, flexible organization technique. This approach requires no change to the structure or content of participating repositories.

Related work

To what extent do existing systems address current problems? There are two broad approaches: distributed file systems and heterogeneous databases.

Distributed file systems provide file access through a network of distributed file servers that may have different architectures. The resulting file space on the server can be extremely large. One problem with these systems is that a file name can have several meanings: it can indicate the file's location in the file system hierarchy; it

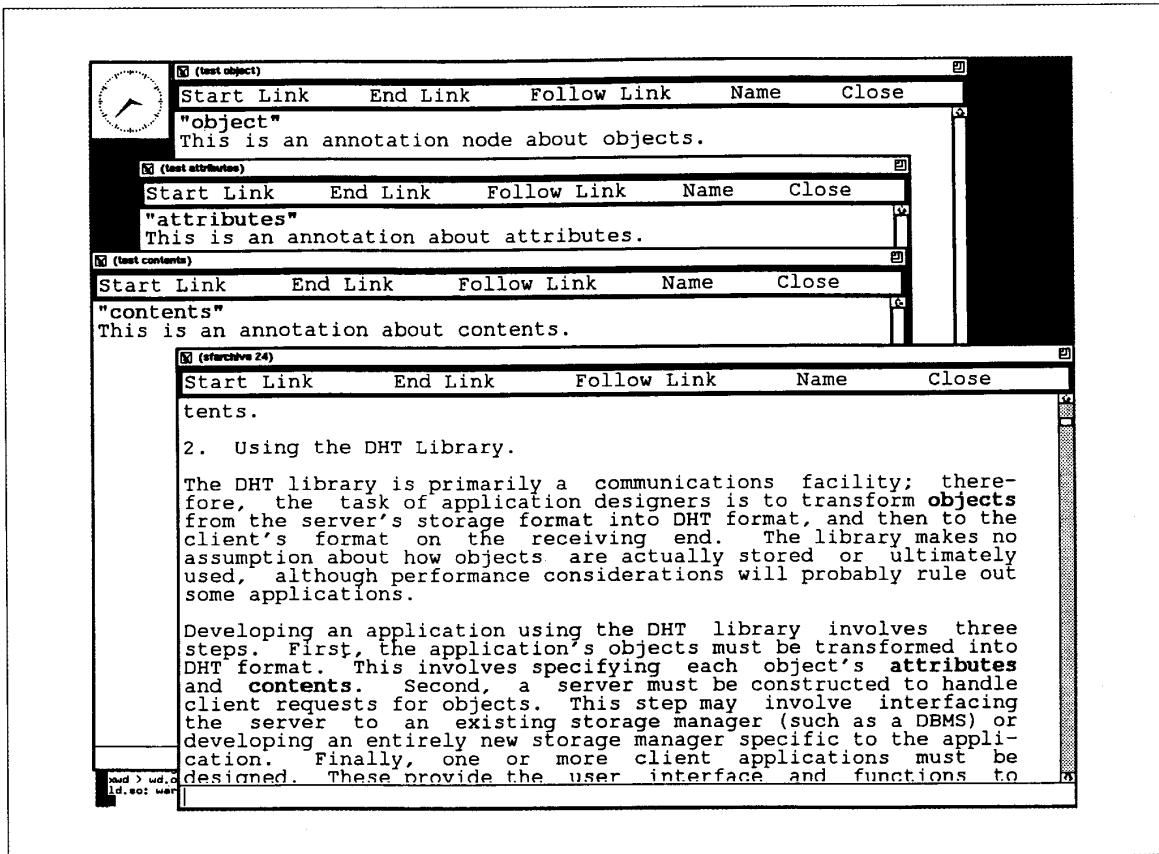


Figure 1. Displaying links.

can express the file's purpose; it often contains file name extensions that indicate the file type; and it may include some notion of the file's relation to other files in the same directory. This overloading can lead to complex, cumbersome file names that are difficult to manage and evolve.

Attributed file systems¹ attempt to solve this problem by attaching attributes to files. These attributes express additional information, such as type, creation and access methods, and relations. In addition, they provide a way of locating relevant files by associative search against attribute values.

All file systems share the drawback that relations among objects can only be expressed by grouping them into directories, or, in attribute-based systems, by attaching attributes that point to other files. Directories can only express membership in a set; attributes force relations among separate entities to be expressed as participation in one or more other objects.

Database management systems address the problem of associating objects by providing primitives that model relations explicitly. Heterogeneous database systems add the ability to integrate many separate databases into logically unified worlds. This can result in a single global schema or in many application-specific schemas, as in federated architectures.²

The chief problem with heterogeneous database systems is preserving autonomy while providing transactions. A transaction is typically controlled by a transaction manager. If the transaction involves objects from more than one database, then some or all participating databases become subordinate to the transaction manager, violating their autonomy.

In summary, file systems offer flexibility and ease of use, but limit organization to hierarchies of directories. Databases offer many high-level features, including powerful organization primitives and multiple views, but sacri-

fice autonomy for transactions. Yet, there is a gap between the two — a gap that DHT can fill.

Distributed hypertext: A solution

Hypertext is a simple concept for organizing and viewing information. It stores chunks of text in objects called *nodes*, which may be individual files, database entries, or possibly text generated on demand at each access. These nodes can be logically connected by using named relations called *links*. Links can represent such concepts as semantic relations between nodes, logical progressions from one node to another, citations in an article, or cross references. They can be anchored, in which case the endpoints of the link are represented by an icon or other indicator in the contents of the linked nodes. For example, Figure 1 highlights the anchors in

the displayed node. Usually, the links are one-way; the resulting structure forms a directed graph called a *corpus*. Users navigate through the corpus by following links from node to node.

Our integration solution is based on the hybrid nature of hypertext. Conklin³ describes the essence of hypertext as a combination of three components:

- a database method, that is, a particular way of accessing information by following links to information nodes;
- a representation scheme, a technique for structuring information; and

- an interface modality, a style of user interaction based on direct manipulation of link buttons.

We exploit these three features of hypertext to achieve integration while preserving autonomy. Specifically, we address issues of organization and flexibility, transparency, and autonomy.

Organization and flexibility. Hypertext offers a simple, natural method for organizing textual data. It can be extended to handle multiple media types, or hypermedia. (Henceforth, we will use the terms *hypertext* and *hypermedia*

interchangeably.) User interaction is simple and straightforward, based on a common command set applying to all object types. Yet the data elements can be organized into complex structures by linking them together. Furthermore, data elements typically have attributes, so the hypertext can be searched by querying against node attributes, links, or contents.

Transparency. Schatz⁴ describes three types of transparency that an information environment must provide:

- type transparency, that is, the ability to use the same interaction technique to manipulate an object independently of its type;
- location transparency, whereby an object should be uniformly accessible, whether it is local or remote; and
- scale transparency, which requires objects to behave the same whether there are 100 or 100,000 in the system.

A heterogeneous environment must also address source transparency: Objects should be accessed uniformly regardless of the type of repository that manages them.

Hypermedia systems have demonstrated type transparency by presenting seamless access to diverse media types, including text, graphics, sound, and images. Distributed hypertext systems have demonstrated location transparency (see the sidebar "Survey of distributed hypertext systems"). It remains to be seen whether hypertext can achieve scale transparency in a very large system containing hundreds of thousands of objects. Our approach is discussed under "An architecture for distributed hypertext."

Autonomy. Hypertext transactions are simpler than those of databases. A hypertext transaction typically involves reading or editing a node, or traversing a link. These operations are performed on a single object: a node or a link. Thus, a single repository manages a single transaction. Because transactions do not cross administrative boundaries, they can be supported without global control. This makes hypertext ideally suited to integrating autonomous information sources.

Hypertext nodes and links can be com-

Survey of distributed hypertext systems

Most hypertext systems are single-user, centralized designs.¹ There are several examples, however, of distributed hypertext systems. Most of them employ a single server to store the links and nodes for a hypertext. Client programs running on workstations connected by a local area network access this server to manipulate portions of the hypertext. Examples include Neptune,² which stores nodes and links in a single database, and Intermedia,³ Document Integration Facility (DIF),⁴ and Knowledge Management System,⁵ which store links in a database and nodes in files.

Multiple server designs allow nodes to reside on several different server machines and links to be made between such nodes. This category includes Sun's Link Service,⁶ Distributed DIF, and Virtual Notebook System,⁷ which store links and node locations in a database, and PlaneText,¹ which stores both links and nodes in Unix files.

Other distributed systems like Telesophy⁸ and those proposed for Xanadu⁹ and Open Hyperdocument Systems¹⁰ are intended to link the knowledge base of a large organization, community, or nation.

References

1. J. Conklin, "Hypertext: An Introduction and Survey," *Computer*, Vol. 20, No. 9, Sept. 1987, pp. 17-41.
2. N.M. Delisle and M.D. Schwartz, "Neptune: A Hypertext System for CAD Applications," *SIGMod Record*, Vol. 15, No. 2, June, 1986, pp. 132-142.
3. N. Yankelovich et al., "Intermedia: The Concept and the Construction of a Seamless Information Environment," *Computer*, Vol. 21, No. 1, Jan. 1988, pp. 81-96.
4. P.K. Garg and W. Scacchi, "A Hypertext System for Software Life Cycle Documents," *IEEE Software*, Vol. 7, No. 3, May 1990, pp. 90-99.
5. R.M. Akscyn, D.L. McCracken, and E.A. Yoder, "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations," *Comm. ACM*, Vol. 31, No. 7, July 1988, pp. 820-835.
6. A. Pearl, "Sun's Link Service: A Protocol for Open Linking," *Proc. Hypertext 89*, ACM, New York, 1989, pp. 137-146.
7. F.M. Shipman III, "Distributed Hypertext for Collaborative Research: The Virtual Notebook System," *Proc. Hypertext 89*, ACM, New York, pp. 29-35.
8. B.R. Schatz, "Telesophy: A System for Manipulating the Knowledge of a Community," *Proc. Globecom 87*, ACM, New York, 1987, pp. 1181-1186.
9. T. Nelson, "Managing Immense Storage," *Byte*, Vol. 13, No. 1, 1988, pp. 225-233.
10. D.C. Englebart, "Knowledge-Domain Interoperability and an Open Hyperdocument System," *Proc. Computer Supported Cooperative Work*, ACM, New York, 1990, pp. 143-156.

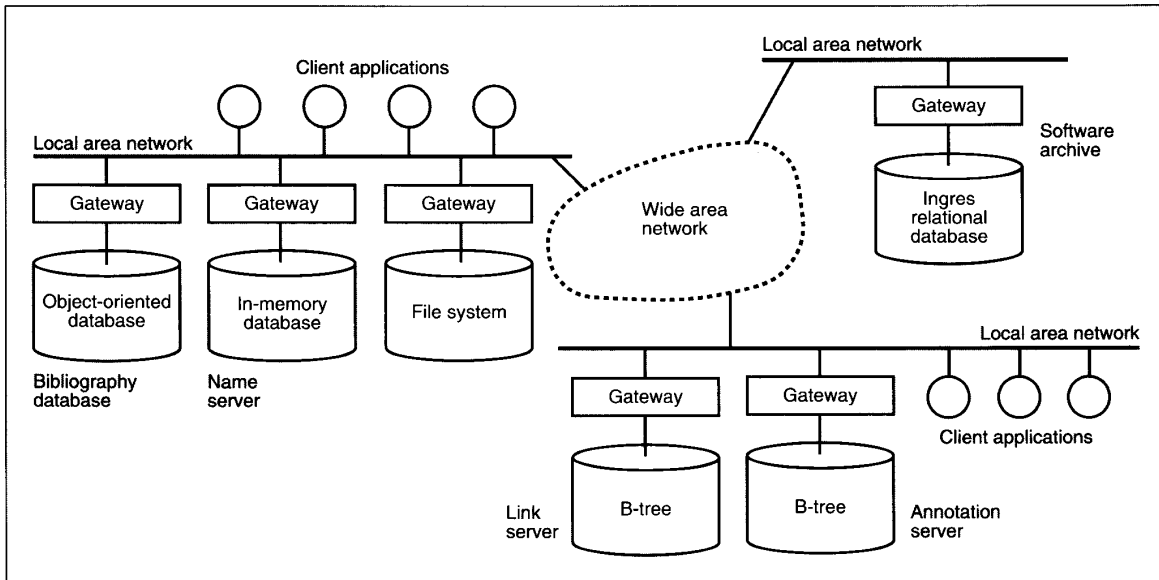


Figure 2. Components of the distributed hypertext architecture.

bined to form complex structures. This capability can be exploited to accommodate *design autonomy*, whereby local databases preserve the ability to maintain their structure and content for existing applications. By providing dynamic gateway processes, local objects can be transformed into hypertext structures, and hypertext operations can be transformed into local queries. The underlying database structure remains unchanged.

An architecture for distributed hypertext

To implement a hypertext solution, we begin with a DHT architecture as the infrastructure for distributed information management. This architecture is based on a client-server model and includes four components: a common hypertext data model, a communication protocol, servers, and client applications. Figure 2 shows these components, which are described below.

Common hypertext data model. From the client point of view, a common model describes all objects in the information space and defines both the structure of information and the operations

allowed on objects. The data model has three basic object components:

- nodes, the container objects;
- links, representing relationships between nodes or sections within nodes; and
- attributes of links or nodes, which identify various object properties.

A set of operations on these components defines how clients access nodes and links:

- create an object;
- update the contents or attributes of a node or the endpoints or attributes of links;
- delete nodes and links; and
- traverse a link from one node to another.

Communication protocol. All components communicate via a common application protocol that implements the operations defined by the data model and provides a mechanism for moving objects between clients and servers.

Servers. The content and structure of the hypertext are managed by servers that include two components:

- a gateway process that transforms hypertext operations into local ac-

cess operations and local information objects into hypertext nodes or links, and

- an information repository that contains the information to be accessed. These repositories are the entities to be integrated; they may be file systems, databases, or special purpose storage managers.

From the repository point of view, the gateway process appears to be another local application that accesses the repository data. In this manner, the DHT architecture can incorporate existing databases without having to copy their data or modify their schemas. Also, existing applications continue to function as before.

Client applications. Clients implement applications and specific styles of user interaction that serve as a user interface to the primitive operations provided by the hypertext data model.

DHT architectural advantages

This architecture solves the integration problem by implementing hypertext in a distributed, heterogeneous environment and taking advantage of

Implementing a server

Servers must perform three tasks to support browsing: export data objects, provide type descriptions, and execute object constructors. Here we describe how we implemented these tasks for our Software Archive server.

The server provides access to an existing Ingres database that manages relations between software project components such as specifications, manuals, and source code. This database has been used to archive student projects collected over the past three years. The archive is partitioned into more than 80 *projects*, each of which is then further partitioned into seven subsections called *forms*. Each form corresponds to a phase in the software lifecycle and contains numerous *modules* that are the actual contents of a project.

The database schema comprises two relations: a *Projects* relation that stores the project's name, file system location, and login names of users having access to the project, and a *Modules* relation that stores information about each module in a project. The figure depicts these relations.

This database, though simple, presents several interesting problems to an integrator. All projects have the same forms, and each form has a set of "core" modules that must always be present. This core set may be augmented by additional modules at the discretion of project team members. Any module added to the database must be associated with a project and a form within that project. These constraints can cause update anomalies if users are allowed unrestricted authority to create views of the database.

DHT solves this problem through object constructors that create components of complex objects in a locally consistent manner. The module constructor, for example, requires that the client specify the "parent" form that is to contain the module, as well as the values for some of the module's attributes. The constructor procedure supplies values for the remaining attributes (status, type, and author) and uses the client-supplied parent argument to determine appropriate values for the project and form fields. The resulting values are appended to the *Modules* relation. However, if the parent object does not

exist or the user is not assigned to the project, the creation fails and an error message is returned.

The process of creating a server for this database involved four steps:

- (1) Deciding the types of nodes and links to be exported and coding the type descriptions.
- (2) Determining which operations to support.
- (3) Writing SQL queries to implement these operations, which will be called from the server process. These include the constructor queries to create new projects and modules.
- (4) Linking these components with the server library and registering the new server with the DHT name server.

There are three basic types to export from the archive database: projects, forms, and modules. The first two are directory objects that serve as the source for links to their children. Modules are container objects that have editable contents. The figure shows the type descriptions for each of these.

The archive server supports the full range of DHT read operations and the creation of project and module objects. Queries implementing these operations must therefore map DHT concepts and objects into SQL queries against record attributes. For example, a "get links" request for a form object is translated into an SQL query that retrieves all modules that are part of the specified object:

```
SELECT DISTINCT project, form, name, instance
FROM modules
WHERE project =:_project
AND form =:_form;
```

Of particular interest is the implementation of the *Module* constructor. From the DHT perspective, this operation creates a new node and links it to the specified form. From the database perspective, it adds a new record to the *modules* table, with the parent field set to the specified form.

the transparency, organization, and flexibility inherent in hypertext, while exploiting its data access modality to preserve repository autonomy.

Heterogeneity and integration. The common data model accomplishes integration by letting client applications interpret diverse information from diverse sources in a uniform manner. A key advantage to this integration strategy is that an information source can participate without having to coordinate with other sources. This is because hypertext nodes are distinct entities and their relations are represented via links that are separate from nodes. As a result, integrating a new repository requires only translation of objects into the common data model and implementation of a server to accept and process

protocol messages. Adding a new server does not affect existing nodes, links, or servers.

Transparency. The hypertext data model describes the structure of objects and the operations on objects; the communication protocol implements these operations. Therefore, any client can manipulate any object using the same set of operations, regardless of source or type.

Transactions and autonomy. Repositories maintain complete control over the type and number of objects exported, who may access them, and what operations may be performed. Only exported objects are known outside the repository; there is no requirement to export a repository's entire schema.

Also, there is no need for a particular server to provide write access to its database, either to users or to system functions. Furthermore, existing applications access the database in the same way, because the server handles all translations between local and common object descriptions.

Transactions are restricted to read or update operations on a single node or link and to create operations that invoke a server's constructor procedure. Therefore, there is no need for a global transaction manager that might infringe on local control and authority, since any transaction is managed solely by the affected server.

The result of this approach is to render a complex, evolving object space comprehensible by applying a simple set of structuring operations.

Relation: Modules			defclass Module {DHT Node}		
Name	Type	Length	{attributes		
project	c	20	{{name	type:string	card: 1 desc: "module name"
form	c	32	{instance	type:int	card: 1 desc: "instance number"
name	c	12	{type	type:string	card: 1 desc: "module type"
instance	integer	4	{heading	type:string	card: 1 desc: "module description"
type	c	12	{status	type:int	card: 1 desc: "module status"
heading	c	64	{author	type:string	card: 1 desc: "module author"}}
author	c	16	{constructor		
status	integer	4	{{name	type:string	card: 1 desc: "module name"
			{instance	type:int	card: 1 desc: "instance number"
			{heading	type:string	card: 1 desc: "module description"
			{contents	type:string	card: 1 desc: "module contents"
			{parent	type:oid	card: 1 desc: "module parent"}}
Relation: Projects			defclass Project {DHT Composite}		
Name	Type	Length	{attributes		
name	c	20	{{name	type:string	card: 1 desc: "project name"
home	text	1024	{home	type:string	card: 1 desc: "root directory"
			{engineers	type:string	card: * desc: "project developers"}}
Relation: Engineers			{constructor		
Name	Type	Length	{{name	type:string	card: 1 desc: "project name"
project	c	20	{home	type:string	card: 1 desc: "root directory"
name	c	16	{engineers	type:string	card: * desc: "project developers"}}
			defclass Form {DHT Composite}		
			{attributes		
			{{name	type:string	card: 1 desc: "Form name"}}
			{constructor {{{}}		

Software archive relations and associated DHT type definitions.

These queries are embedded into a set of C language functions that are linked into a server template provided as part of the DHT library. Once this process is complete, the new server is ready to run. The last step is to add its name to the name

server so that its address is registered upon startup.

The entire development process for this server was completed in two days and the implementation comprises about 300 lines of C and SQL code.

Implementation

We have developed a prototype DHT system implementation. Here we present an overview of the system to describe how hypertext functions are carried out from the client and server points of view.

System overview. The goal of our prototype system is to demonstrate how existing heterogeneous repositories can be integrated using the DHT architecture. Therefore, we built several servers to provide access to a variety of repositories ranging from simple hash tables to object-oriented databases. Specifically, servers were constructed to access an existing software project database implemented using the Ingres relational database management system, an ob-

ject-oriented database containing bibliographic entries, a name server to manage addresses of the other servers in the system, and the Unix file system.

Additionally, we implemented link and annotation servers using a B-tree storage manager. The link and annotation servers store annotations linked to other objects, as well as user's personal links. These servers enable users to attach comments to other nodes in the system and to build personal link networks. We have several clients that provide user access to the available servers, including a hypertext browser providing a means for navigating links, viewing and editing nodes, and creating annotations to nodes, and several shell commands for creating and retrieving nodes from specific servers.

See Figure 2 for an overview of these

components and the sidebar for details on implementing a DHT server.

Using the system. Users engage in three basic activities: browsing links, viewing and editing nodes, and creating new objects. These tasks are carried out using a window-oriented browser/editor. Servers support these activities by exporting data objects, providing object type descriptions, and creating new objects in their storage managers in response to user (or client) requests. We examine each of these activities.

Browsing. Browsing involves repeated execution of the following steps by a client process:

- (1) Locate appropriate link servers. As discussed previously, links are not

necessarily stored as part of the nodes that they link. Therefore, a client managing the browsing process must first determine what server to contact to retrieve the list of links originating at a node. This server may be specified with each command or in a default configuration file.

By convention, each user has a default configuration that specifies, among other things, where to look for links.

(2) Retrieve the list of links associated with a node. Once the link server has been specified, the client simply sends a message to the server requesting the list of links emanating from the desired node.

(3) Filter the list according to attributes specified by the user. Applications and users may be interested in only a subset of links associated with a node. Therefore, a user may specify a filter to apply to the list of links.

(4) Present the resulting links to the user. This step is application dependent as well as link dependent. Some links may simply represent a relationship between two nodes; others are "anchored," representing a relation between a subcomponent or section of one node to a subcomponent or section of another node. Anchored links are best presented by emphasizing the "anchor" in the displayed node, as in Figure 1 where anchors are highlighted, while other links can be selected from a menu-style list.

(5) Retrieve and display the endpoint of a chosen link. Once the links have been presented, either as highlighted anchors or menu items, the user can select one for traversal. This step can take place only after a client determines that it has sufficient local resources to display a node. This is because certain node types require special hardware or software; a video object, for example, requires special playback and display hardware whereas text can be shown on any terminal. To facilitate this step, object identifiers contain a type field, so clients know an object's type before retrieving the whole object. Once the client process determines that it can present the contents of a node, it sends a message to the server that stores the node, requesting the return of the contents.

Creating new objects. Servers must have absolute control over the creation process to ensure consistency of the local repository. This requirement complicates object (node or link) creation.

Hypertext combines a user interaction technique, a data representation method, and a data storage mechanism.

It means that users cannot arbitrarily create nodes or links at any server. Rather, to allow servers to control access and meet repository consistency constraints, we provide object "constructors," local procedures that create complex objects from client-supplied arguments.

Constructor arguments are specified in object type descriptions. Users who want to create an object must specify the type of object to create. The client then retrieves the appropriate type description and interprets the constructor argument specification. The user is then asked to supply values for each of the constructor arguments. The figure in the sidebar shows a type description with constructor arguments.

We began with a statement of four requirements for information integration: organization, flexibility, transparency, and autonomy. Our DHT architecture meets these requirements by extending hypertext to a distributed, heterogeneous environment.

We chose hypertext as a solution strategy because of its multifaceted nature: it combines a user interaction technique, a data representation method, and a data storage mechanism.³

The user interaction facet provides transparency. Users interact with information by creating nodes and links and by browsing the resulting linked information space. Browsing utilizes a common set of tools regardless of the types of nodes contained in the information space.

The data representation facet provides the organization and flexibility to construct multiple views and complex structures by linking nodes. This simple

organization technique is both powerful and extremely flexible. Views can be changed or removed entirely simply by adding or removing links. Yet the linked nodes remain unaffected by such changes. Contrast this with database management systems, in which different views can produce critical update problems and changes in schema structure can affect numerous data instances.

Finally, the data storage facet provides the key to implementing an integrated hypertext in an autonomous and heterogeneous environment. This is because the hypertext data storage model can support transactions without compromising the autonomy of participating information repositories. Furthermore, a hypertext storage mechanism can manage diverse data types and can be implemented on a variety of storage managers.

Thus, the hypertext-based approach results in two gains: a simple integration strategy that preserves repository autonomy and a powerful organizational tool that lets many users and applications construct personal views of shared objects.

The chief weakness of our solution is the necessity to implement a new gateway for each new repository; this is a weakness we share with other integration strategies. However, our experience has shown that new gateways can be implemented in as little time as one day, so this is less of a drawback than it might appear.

Our future work will focus on developing reusable mechanisms for integrating new servers as well as continuing to gather experience with the prototype system. We are also implementing more sophisticated clients that model software processes⁵ and servers to integrate repositories for software engineering environments. ■

Acknowledgments

Christina Chen, Hongyan Luo, and Vien Chan participated in the implementation of the prototype system. We thank Peter Danzig, Deborah Estrin, Doug Fang, Shahram Ghandeharizadeh, Steve Hotz, and Kim Korner for their many helpful comments. This work has been supported in part by contracts and grants from AT&T, Northrop, Pacific Bell, and the Office of Naval Technology through the Naval Ocean Systems Center. No endorsement is implied.

References

1. M. Theimer, L.F. Cabrera, and J. Wyllie, "QuickSilver Support for Access to Data in Large, Geographically Dispersed Systems," *Ninth Int'l Conf. Distributed Computing Systems*, CS Press, Los Alamitos, Calif., Order No. 1953, 1989, pp. 28-35.
2. D. Heimburger and D. McLeod, "A Federated Architecture for Information Management," *ACM Trans. on Office Information Systems*, Vol. 3, No. 3, July 1985, pp. 253-278.
3. J. Conklin, "Hypertext: An Introduction and Survey," *Computer*, Vol. 20, No. 9, Sept. 1987, pp. 17-41.
4. B.R. Schatz, "Telesophy: A System for Manipulating the Knowledge of a Community," *Proc. Globecom 87*, ACM New York, 1987, pp. 1181-1186.
5. P. Garg and W. Scacchi, "ISHYS: Designing an Intelligent Software Hypertext System," *IEEE Expert*, Vol. 4, No. 3, Fall 1989, pp. 52-64.



John Noll is a doctoral student in computer science at University of Southern California. He received a BA in physics from Colorado College in 1979, a BS in industrial and systems engineering from USC in 1980, and an MS in computer science from USC in 1990. His research interests include distributed hypermedia systems, wide area networks, and large-scale software engineering.



Walt Scacchi is an associate research professor in the Decision Systems Department at the University of Southern California. Since joining the USC faculty in 1981, he created and now directs the USC Systems Factory Project, the first software factory research project in a US university. His research interests include very large scale software production, knowledge-based systems for modeling and simulating organizational processes and operations, CASE technologies for developing large heterogeneous information systems, and organizational analysis of system development projects.

Scacchi received a BA in mathematics and a BS in computer science in 1974 at California State University, Fullerton, and a PhD in information and computer science at University of California, Irvine in 1981. He is a member of the ACM, IEEE, AAAI, Computing Professionals for Social Responsibility, and Society for the History of Technology.

Readers can contact the authors at the University of Southern California, Bridge Hall 401V, Los Angeles, CA 90089-1421 or at their e-mail addresses: jnoll@pollux.usc.edu and scacchi@pollux.usc.edu.

December 1991

Command a Leading Edge in Computer Research

We are recruiting the best talent available. So that we can scale new heights in computer research and command a leadership position in the Pacific region. With close to 100 research staff, we are currently looking for new talent. If you are in the relevant disciplines, consider seriously a future with one of the top computer research laboratories in the region.

Our objectives are two-fold: a) To carry out world-class research and b) To carry out significant technology transfer. The research focus of the Institute broadly covers three strategic areas: **Multimedia, Artificial Intelligence and Natural Language Processing.**

Specific projects cover Virtual Environments, Video Classification, Image Processing, Scientific Visualization, Text Retrieval, Neural Networks, Fuzzy Logic, Computer-Aided Translation, High Speed Networks, Distributed Computing and Parallel Computing. The Institute has embarked on a number of joint projects with industry in Singapore and overseas. These include Computer Aided Translation with IBM; Pattern Recognition with the Port of Singapore Authority; Connectionist Expert System Shell with Singapore Airlines; and Text Abstraction with the Ministry of Defence.

To qualify, you should have a PhD, Masters or Bachelor's in computer science, electrical engineering and related disciplines, and preferably experience in R&D laboratory work. *(And if you're a Singaporean or come from Asia, all the more reason for you to consider coming home.)*

We offer:

- A stimulating work environment in one of the most advanced computing facilities in the region
- A competitive salary
- Attractive fringe benefits

If you're ready to take up this top-notch research opportunity, please send your resume to the **Director of Personnel, National University of Singapore, 10 Kent Ridge Crescent, Singapore 0511** or fax c/o **ISS Recruitment Manager at 775-0938** or **BITNET ISSAPPLY@NUSVM.**

INSTITUTE OF SYSTEMS SCIENCE



NATIONAL UNIVERSITY OF SINGAPORE