A Distributed Graph Storage System for Artifacts in Group Collaborations

TR92-010
March, 1992

Douglas E. Shackelford
John B. Smith
F. Donelson Smith

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175
919-962-1792
jbs@cs.unc.edu

A TextLab/Collaboratory Report

Portions of this work were spported by the National Science Foundation (Grant #IRI-9015443 and by IBM Corporation (SUR Agreement #866).

0. Abstract

Our group is building an artifact-based collaboration support system and studying the collaborative process. This paper discusses the data storage component of the system. It supports a graph-based data model, conservatively extended to meet hypermedia requirements. It is implemented in a distributed architecture so data may be stored in multiple locations and moved among locations . Issues addressed in both the paper and the system include: scale, logical and physical partitioning, protection, concurrency, and support for an extensible set of user applications. The discussion emphasizes issues and system aspects concerned with collaboration and support of multiple concurrent users.

1. Introduction and Motivation

Our research focuses on the process of collaboration and on technology to support that process. We are concerned with intellectual collaboration required for designing software systems and other similar tasks in which groups of people work together to build large, complex structures of ideas. The work of such groups -- either directly or indirectly -- is concerned with producing some tangible artifact.

> The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed. I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. [Brooks, 1987]

According to Brooks, the fundamental problem in software development is building the large "conceptual construct." To facilitate communication among group members that create the construct, groups generate a variety of interrelated elements of the artifact. For software systems, the artifact may include concept papers, architecture, requirements, specifications, programs, diagrams, reference and user manuals, as well as administrative documents; for other tasks, the artifact may contain these and/or other kinds of information. The artifact is created by the group both for its own use and as part of the final product. Our research in the UNC Collaboratory project studies how groups merge their ideas and their efforts to build an artifact, and we are developing a computer system (called ABC for Artifact-Based Collaboration) [Smith & Smith, 1991] to support that process.

In the following sections we discuss the storage system for group artifacts being developed as part of the ABC system. The storage system has a distributed implementation and is called the *ABC Distributed Graph Storage System (ABC/DGS*, or just *DGS* when the reference is clear from the context). Section 2 gives key requirements for the system; section 3 describes the data model; section 4 describes group-related issues; section 5 sketches the system implementation; section 6 describes current status; section 7 relates our design to other work, and section 8 gives a summary and conclusions.

2. Requirements for Storing Group Artifacts

In this section we give a brief summary (in no particular order) of key requirements that have shaped our storage system design.

Permanent (persistent) storage -- obvious but fundamental.

Represent structural and semantic relationships -- all data elements that comprise the artifact have implicit structural and semantic relationships. For example, structural relationships in a document show ordering among chapters, sections, and paragraphs; semantic relationships can link an idea introduced in a concept paper to its description in requirement and design documents, to its implementation in a program, and to an explanation in a users' manual. Our requirement is to make such relationships explicit to aid in locating information and in maintaining coherence, completeness, and correctness of the materials. For this reason, functions provided in the storage system should support hypermedia applications such as browsers and other navigation aids [Haan, 1992].

Comprehensible organization -- as artifact size increases and relationships become many and complex, users can lose their orientation and become "lost in hyperspace" [Halasz, 1987]. To avoid this condition, users must be able to isolate small, coherent portions of a large artifact. Once the artifact has been organized into smaller structures, the parts can be understood more easily and then related to each other via semantic links.

Sharing with protection -- because the artifact effectively constitutes a form of collective memory for a collaborative group, it must be sharable by all. There are, however, good reasons to make it possible to authorize or deny access to selected elements of the artifact by individuals or sub-groups.

Private data -- these are created by individuals for their own use. Examples include personal notes, annotations on documents, and correspondence. Users should be able to create and protect such data and still establish relationships among them and the public artifact.

Concurrent access -- since collaborators must work together, it is often necessary for more than one user to read or modify some part of the artifact at the same time. Data consistency semantics in these cases should be easily understood and provide minimal barriers to users' access to the artifact.

Responsive performance -- sufficient to support interactive browsing of the artifact is required.

Scalable -- we are concerned about scale in two respects: the number of users in a group (and consequent size and complexity of the artifact), and the geographic dispersion of group members. For small software design teams (5-10 people) working over months or years, we estimate that artifacts comprised of O(10,000) elements will be needed. A system that can support industrial software development will require at least two orders of magnitude greater capacity, and perhaps more for defense, aerospace, and other large systems efforts [Malcolm, 1991]. To be scalable, it must be possible to distribute the system over available processing and network resources and to add resources incrementally as necessary. To achieve this distribution, users should be encouraged to organize the artifact into manageable elements. This is also true from the standpoint of human comprehension as well as capacity and performance.

Available -- if data becomes unavailable because of system faults, users may be severely impacted. The system must, therefore, be designed to tolerate most common faults and continue to provide access to most or all elements of the artifact. Replication of data and processing capacity will be required to achieve high availability.

User and artifact mobility -- users will need to change locations and system administrators will need to move data or processing resources to balance loads and capacity. The system should support this mobility in a way that is transparent to users and application programs. There should be no location dependencies inherent in the storage system.

Support for applications -- many applications used by a group are likely to be existing tools such as editors, drawing packages, compilers, and utilities, which use a conventional file model for persistent storage. The system should make it possible to use such tools with minimal or no changes. New applications developed for use with the system should not be dependent on any particular platform for implementation.

3. Data Model

Our data model addresses the artifact storage requirements of groups collaborating to create complex constructs, especially groups designing software systems. The data model is especially well suited for conventional hypermedia applications (e.g., navigational browsers), but it also

supports new approaches to applications such as document formatting and printing, make, and version control.

The most basic element of artifact storage is a *node*. Nodes are repositories for information stored either as node *attributes* or as a node *content* variable. Node attributes are named variables of arbitrary type and size. Some attributes (such as creation time and size) are maintained automatically by the system. There may also be an arbitrary number of application-defined and maintained attributes. Node content is used in two ways. First, a node can contain any data represented as a stream of bytes (the familiar model used in conventional file systems). For example, a node's data content could be text, bitmap, line drawing, digitized audio and video, spreadsheet, or any other data. Applications that read and write conventional files can read and write node data in our system with no changes. Second, a node can contain more structured forms of data, which will be described in the paragraphs that follow.

Abstractions for grouping related nodes and composing them hierarchically are essential for managing large artifacts. In our model, nodes are grouped into named collections and these collections are stored as the content of some node (thus, the value of a node's content variable is either a collection of nodes or an arbitrary stream of bytes as described above). This recursion provides a simple but powerful model for composing a complex artifact from smaller elements. It is not, however, enough.

Many essential relationships among parts of an artifact are structural, especially those that indicate access order (e.g., if a group of nodes store parts of a document, it is necessary to represent the structural relationships of sub-sections to sections, sections to chapters, and chapters to the document). A natural expression of structural relationships is a *graph*. For this reason, all collections of nodes are really graphs -- a named set of nodes and links (edges). To reinforce the essential role of composition, the only way a graph can be created is to make it the content of a node and every node must be contained in a graph.

A link in a graph represents a structural relationship between two nodes. Because we consider structure in artifacts to be very important, we adopt the terms *structural-link* (abbreviated *S-link*) and *structural graph (S-graph)*. A common case, however, is an S-graph containing nodes but no links; it represents a set of nodes having non-structural relationships. Nodes and S-links can be contained in more than one S-graph simultaneously but must be contained in at least one. Nodes may have arbitrary numbers of in-coming and out-going S-links. S-links have a direction, although traversal is supported in either direction. Like nodes, S-links can be repositories of information stored in attributes and a content variable.

S-graphs are *strongly typed* in terms of a set of predefined graph types. The current data model includes five types: *general directed graphs, connected graphs, acyclic connected graphs, trees,* and *lists*. The system will guarantee that typed S-graphs are always in a state consistent with their type. No operations are permitted that would violate the integrity of the type. For example, an application is not allowed to create a cycle in an S-graph of type tree. Typed S-graphs are necessary to address issues such as integrity, consistency, and completeness of the artifact.

The value of these rather abstract mechanisms is best motivated by examples. The data model encourages users to decompose a large artifact into small S-graphs related by composition. For example, Figure 1 shows how a software system could be structured. This organization improves human comprehension and increases potential for concurrent access to the program's components. Many common structures, such as documents, can be represented by trees. Unfortunately, large documents result in large trees, making it difficult to visualize and browse the document or to allow multiple users concurrent access for making revisions. Figure 2 illustrates a partitioned document structure and Figure 3 shows a more complex example -- organizing public and private parts of an artifact.

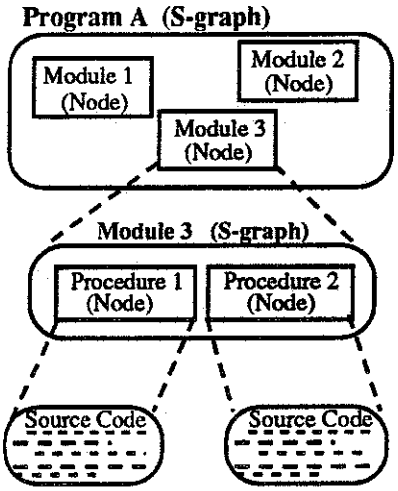

Figure 1: Using Node Content to Represent a Software System

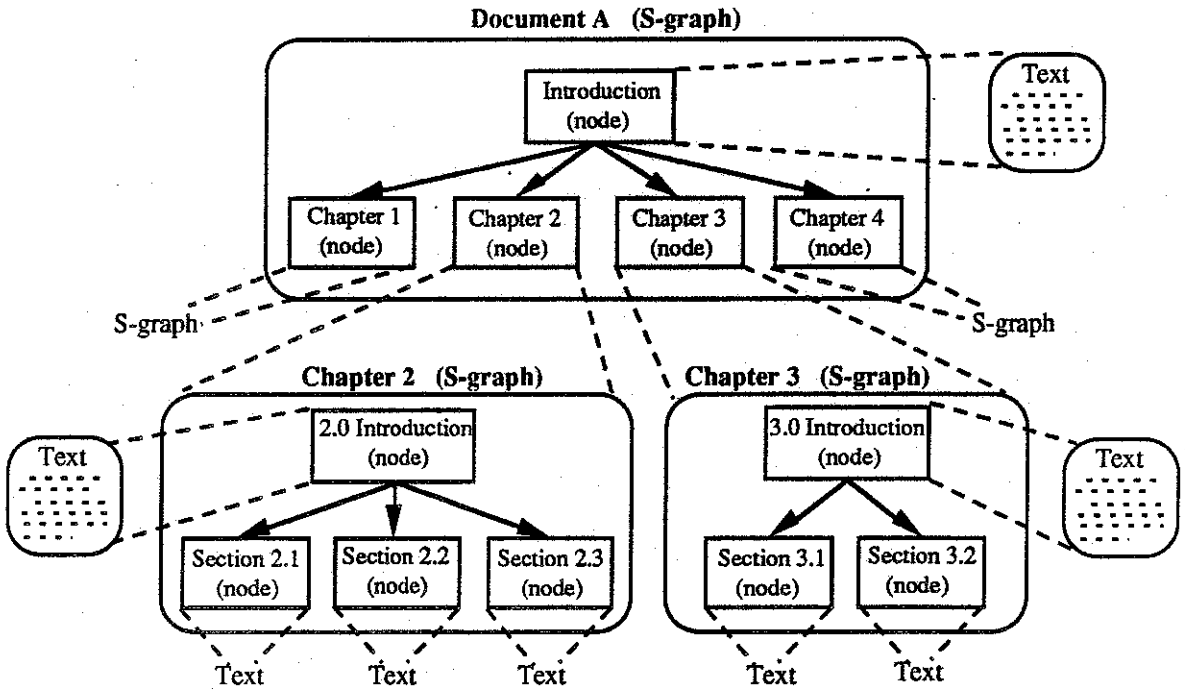

Figure 2: Using Node Content and Structure to Represent a Document

While composition and structure are necessary for organizing complex artifacts, they are not sufficient -- many useful relationships among parts of the artifact cannot be modeled as structure. Some examples are: references in a document to glossary entries, figures, or related sections; private annotations made by a reader but not intended to be part of the document; declarations for classes referenced in an object-oriented program; and references in a specification document to a requirements document. In each case, the relationship cuts across normal structural

boundaries. To express these relationships, we define a more flexible kind of link, called a *hyperlink (H-link)*, that 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 (see Figure 4). Links similar in function to H-links are usually the key elements of conventional hypertext systems. H-links and the nodes they link are grouped into *hypergraphs (H-graphs)*. An H-graph is a set of H-links and nodes such that the set of nodes is exactly equal to the set of all source and target nodes for the H-links. There are no type constraints on H-graphs.

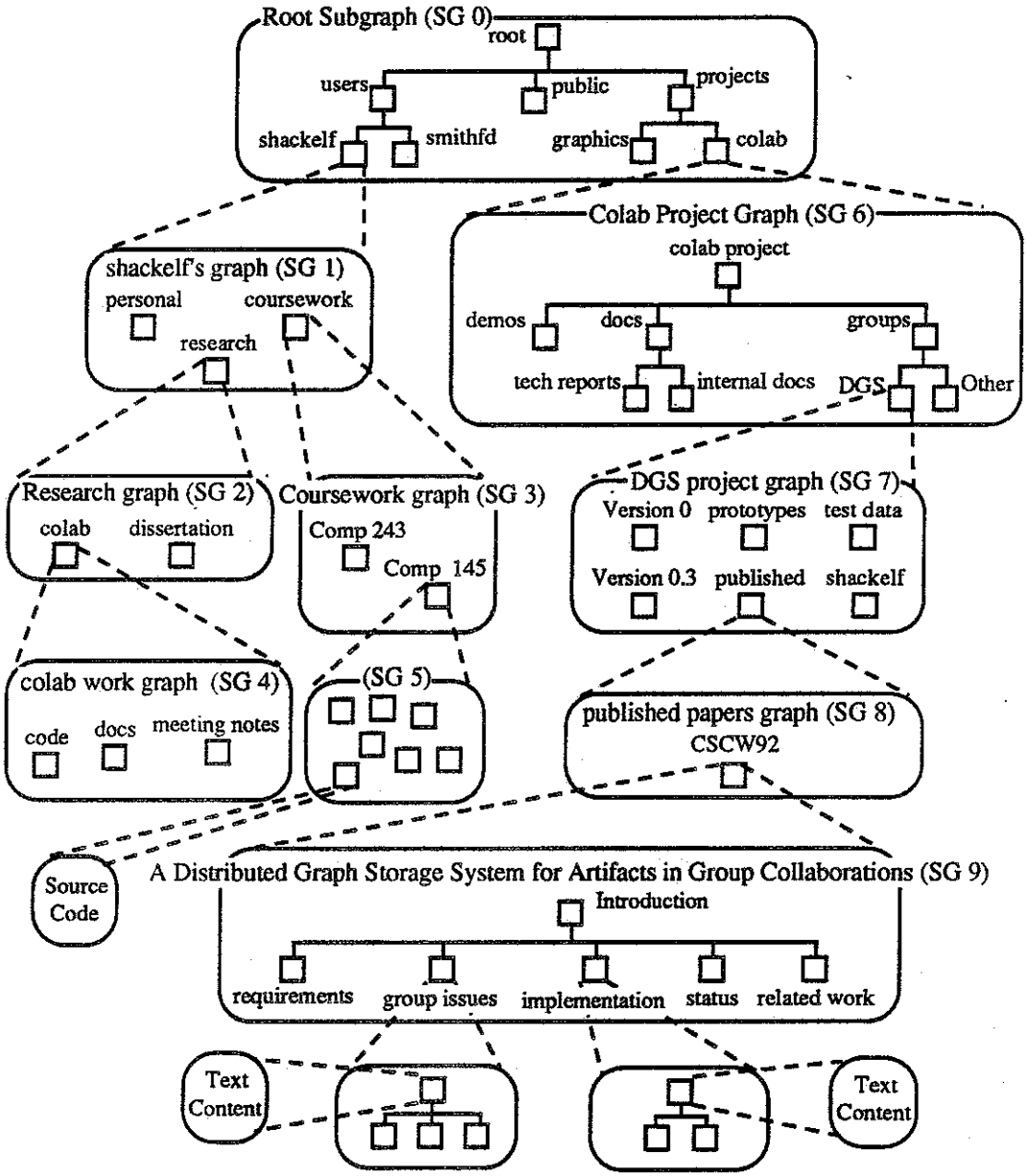

Figure 3: Organizing Public and Private Parts of an Artifact

H-links alone are often not sufficiently precise. For example, a group of users might want to use a node to store the glossary of terms common to their project. It would be desirable to create an H-link from the occurrence of a term in a document node to its definition in the glossary node. Unfortunately, an H-link can link the two nodes which contain the term and the glossary, but it cannot link the term itself to its definition.

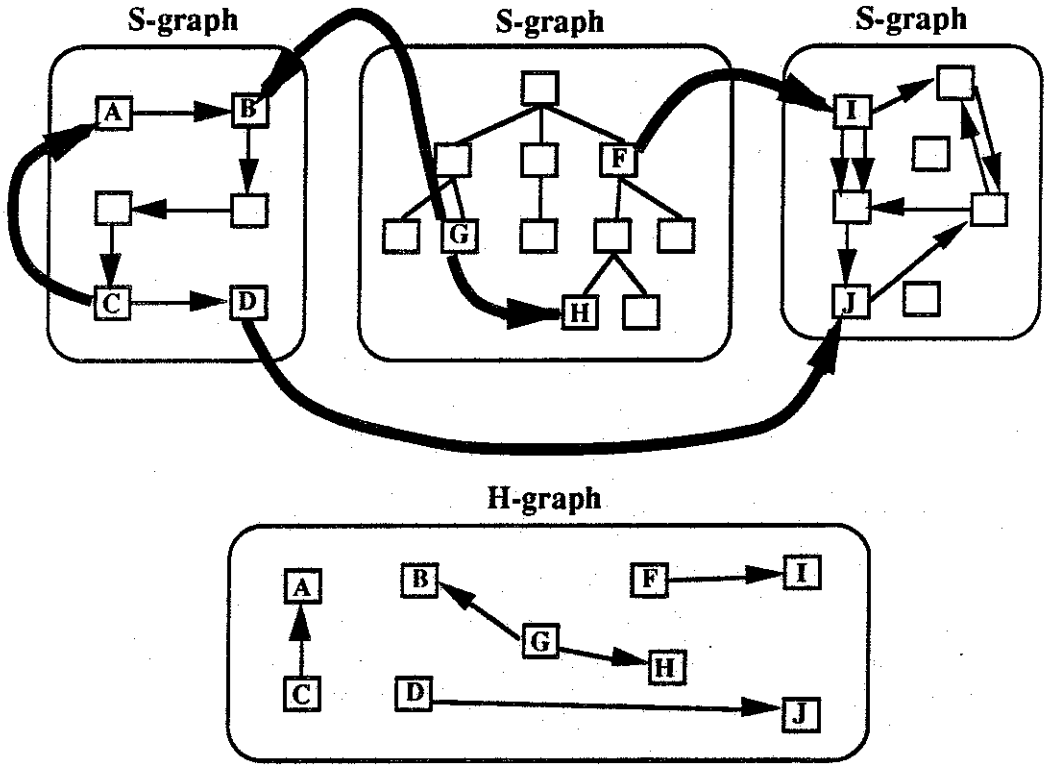

Figure 4: H-Links and H-Graph

To achieve finer-grained H-links, the data model provides the concept of an *anchor* within a node. An anchor identifies part of a node's content, such as a function declaration in a program module, a definition in a glossary text, or an element of a line drawing. An anchor can be used to focus an H-link onto a specific place within the content of a node. An anchored H-link is one which is paired with one or more anchors in its source or target nodes. H-links can be anchored in their source node, in their target node, in both, or in neither. Furthermore, an H-link can be paired with more than one anchor within the same node and several H-links can be associated with one anchor (see Figure 5). Applications are responsible for maintaining an anchor's value so it always identifies the same part of the node content even as the content changes.

In the remainder of this paper, the terms *link* and *graph* are used when the discussion applies equally to S- or H- objects of these types.

Finally, the data model includes *attributes* and *values* that are associated with nodes, links, and graphs. Some attribute variables for nodes and links are called *common attributes* because their values are the same in all contexts, i.e., the attribute value is the same no matter how many graphs contain the node or link (common attributes can also be defined for graphs). Nodes and links also have context-sensitive attributes that are meaningful only in the context of one particular

graph that contains the node or link; these are called *graph attributes*. For example, consider a node contained in both a tree S-graph and a list S-graph. A tree browsing application would store the position of the node within the tree in a graph attribute for the tree S-graph in terms of its relationship to its parent and sibling(s), while a list browser would store the position of the node in a graph attribute for the list S-graph in terms of its predecessor and successor nodes. Graph attributes have proven very useful for maintaining such context-dependent information for nodes that are members of multiple graphs.

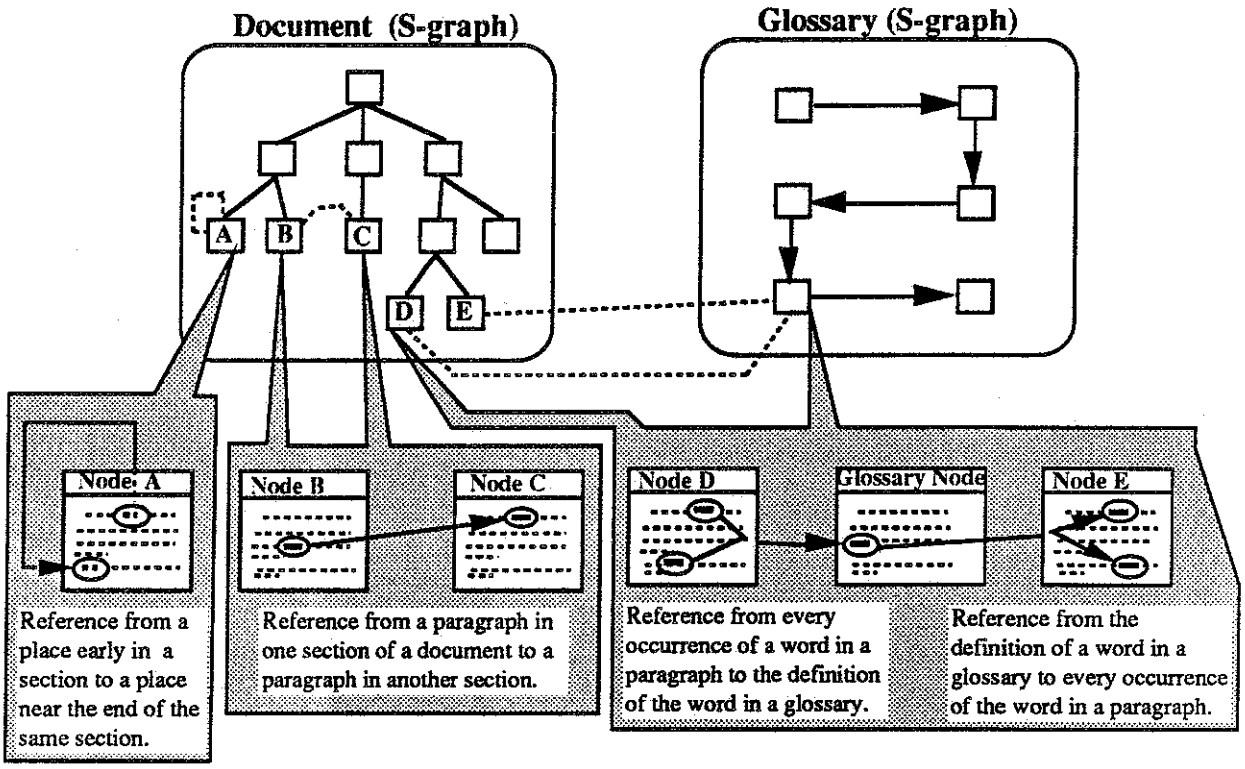

Figure 5: Examples of anchored hyperlinks

All node, link, and graph objects are identified by a 96-bit *object identifier (OID)* that is universal and unique. Once an object is created, its OID is never changed and the value is never reused even if the object is deleted. An OID is treated as an "opaque" (uninterpreted) value by applications and users. An important implication of maintaining an unique, unchanging OID can be seen by considering the semantics of copying objects. Two copy operations for objects are supported -- copy by reference and copy by value. Copy by reference just makes a copy of an object's OID; copy by value creates an exact duplicate of an object and creates a new OID for the second object. Copy by value has the side effect that any existing links and graphs refer to the original object, not the new copy.

4. Group-Related Design Issues

In the previous section we showed how the data model encourages and supports composition of large artifacts from smaller components. Since small graphs are easier to comprehend and share, the model encourages users to group their nodes into many small graphs rather than a few large ones. Partitioning the artifact improves the storage system's ability to scale up to support larger groups by adding capacity incrementally. Moreover, opportunity for concurrent access to objects by multiple users is increased by finer granularity. Performance is also improved when data can be parceled out for storage near people who need it. This is especially important when groups are widely distributed geographically.

Users occasionally need to access parts of a shared artifact concurrently. A major issue is defining appropriate semantics for overlapping operations by two or more users' applications. We expect applications will read attribute and content values much more frequently than they will write them (in the following discussion,we refer to a user's application that changes an object as a *writer* and one that only reads as a *reader*). Given strong support for an artifact composed of small elements, we expect multiple concurrent writers of an object to be rare. Multiple concurrent readers of an object, however, will be common and, furthermore, readers will often need to create anchors in node content (e.g., for private annotations or reference links). To create valid anchors, an application must be processing the most recent version of a node's content. If a reader is allowed to create anchored H-links while a concurrent writer is changing the content of the same node, the new anchor values could be incorrect. We have adopted an approach that allows multiple concurrent readers to create anchors, but only when there is no concurrent writer.

To specify allowable concurrent accesses, we define access modes that determine the operations that are allowed on a node, link, or graph. Read access allows operations that do not change graph membership, linking information, or attribute or content values. To support the special case of reading and creating anchored H-links, *read* access to a node allows anchor creation and deletion. *Read-no-anchor* access to a node allows all operations of read access except anchor creation. *Read_write* access allows all operations. Before a user's application can access a node, link, or graph object, it must explicitly open that object in one of the three access modes.

We can now specify rules for concurrent opens of a single object:

- For links and graphs, multiple opens with *read* access and a single open with *read_write* access are allowed (as is the weaker case of multiple *read* opens alone).
- For nodes, multiple opens with *read_no_anchor* access and a single open with *read_write* access are allowed (as is the weaker case of multiple *read* and/or *read_no_anchor* opens alone).
- No other cases of opens for concurrent access are allowed.

Changes to an object are not visible to any applications with overlapping opens of the object until it is closed by the writer and then only to applications that open it after the close completes. Because graphs and nodes are opened independently, the system encourages browsers that read structure and semantic relationships (in graphs) and applications that write content (in nodes) and vice-versa. The greatest concurrency among applications and browsers is achieved in these cases.

An ideal storage system for group artifacts would create the abstraction of a central, unique copy of each object which can be shared by all users simultaneously (an arbitrary number of concurrent readers and writers). To maintain this abstraction, each update should be synchronized with all viewers of the object. This capability is an open research problem and, consequently, the

first version of our system provides a much weaker guarantee. Future versions will approximate the single copy illusion with increasing fidelity.

Groups can control access to parts of the artifact by specifying *access authorizations* for node, link, and graph objects. Authorizations are expressed in an access control list stored with each object. An access control list maps names of users or groups of users to categories of operations they are allowed to perform on the associated object. No user is allowed to access an object unless that user has proper authorization for operations implied by the access mode specified on open. Access authorizations are given in Table 1. In addition to access authorizations, users can have *administrate authorizations* for objects. A user with administrate authorization can perform operations such as changing the object's access control list.

Object Type	Authorizations and What They Provide	
	read	read_write
node	read anchors, node attributes, and content	change anchor values, node attributes, and content
link	read link attributes and content	change link attributes and content; create/delete anchors for the link
graph	read common attributes, graph attributes of nodes and links in the graph, structure of graph	change common attributes, graph attributes of nodes and links in the graph, structure of graph

Table 1: Authorizations and what they provide.

Private views are created by individuals or small sub-groups for their own uses such as creating annotations or personal reference links in the shared artifact. Private views should not interfere with or clutter the public view of the artifact. Private views are especially important with respect to anchored H-links because a node may have more anchors and links than many users want to see. In addition, S-graph owners may want to restrict the set of users who have authorization to create publicly viewable H-links to their S-graphs. H-graphs are the primary mechanism for distinguishing public views from private views. By so grouping links and using access authorizations on graphs, users can establish desired levels of control over views of links and anchors.

5. Implementation Design

In this section we discuss the implementation of the storage system with emphasis on key design decisions. A fundamental decision was to optimize for fast response to the most frequent application requests. For hypermedia navigational-style browsers that provide the user interface to the artifact, we expect most requests will represent simple queries with bounded scope (i.e., related to a single node or link, or to members of one graph). Some examples are:

- What are all the links from node 6 in H-graph 50?
- What is the value of attribute "XY position" of node 10 in S-graph 100?
- What are the sibling nodes of node 25 in S-graph 100?

More complex queries involving many graphs can also be used but performance may be considerably less responsive. Content search is not currently supported.

Given the expected composition of the artifact from small elements and anticipated modes of interaction through browsers, we believe many characteristics and access patterns of objects will strongly resemble those observed in distributed file systems supporting software teams using workstations [Baker, et. al., 1991]. Our working hypothesis is that an effective implementation can be achieved by applying design ideas such as local caching, bulk-data transfer, and minimal client-server interactions pioneered in high-performance, scalable file systems such as Andrew [Howard, et. al., 1988] and Sprite [Nelson, et.al., 1988].

The basic structure of the system is shown in Figure 6. An application process acts on behalf of a user to read and modify objects. Each host machine runs a single graph-cache manager process that services all applications running on that machine. Application requests are directed over local interprocess communication facilities to the graph-cache manager. The graph-cache manager maintains a local copy of node, link, and graph objects used by application processes and is responsible for implementing all graph operations except for anchor table merging. It is also responsible for maintaining the consistency of typed S-graphs.

When an application opens an object, the graph-cache manager, in turn, opens the object at the storage server and retrieves it using a simple file-oriented protocol. The received object is converted from its representation in a file to a representation designed for fast access in memory. As the application makes requests, the graph-cache manager performs those operations on its local copy. Write operations are reflected in the storage server only when the graph-cache manager closes the object and returns the file representation to the storage server. Each file retrieved from the storage server contains either a whole node, a whole graph, or a group of links. The structure of each type of file is shown in Figure 7. Nodes and graphs are stored individually, whereas links are grouped according to the graph in which they were created.

Storage server processes are responsible for permanent storage of data on disk. The file-oriented interface to the storage server is designed to isolate it as much as possible from the representation and semantics of objects. The primary responsibility of the storage server, therefore, is to store and control access to files indexed by an object's OID. Storage servers also provide services for creating unique OIDs and anchor IDs, and for merging anchor table information created by concurrent readers of the same node.

The storage server must perform several checks before completing an open request. First, it must determine whether the user who is running the application has the correct authorizations to open the object in the requested access mode. Then, the storage server must determine whether the requested access mode is in conflict with any overlapping opens. An open request will fail if the user lacks proper access authorization or if the open conflicts with other opens in progress.

Figure 8 shows a more complete view of the system structure with multiple clients and servers, including servers that provide protection services and mappings from an OID to the host system that is the custodian for that object. Object location is based on dividing the artifact store into non-overlapping collections of nodes, links, and graphs called *partitions*. Partitions form boundaries for administrative controls such as space quotas, load balancing among servers, and replication of data. The *partition number* of an object is embedded in its OID but this substructure is never made visible outside the storage service. An object must remain in the same partition for its entire lifetime because its OID cannot be changed. We distinguish the partition number of an object from its absolute physical location(s) and, by introducing a level of indirection, it is possible to change the physical location of objects while preserving all link and composition relationships with other objects (see Figure 9). We expect, however, that in most cases one storage server will maintain both the partition directory and data storage for an object.

6. Current Status

An initial prototype of the storage system using a database-oriented design was implemented in Smalltalk over a year ago. Experiences from constructing and using this prototype were invaluable in refining the requirements, data model, and programming interface. The performance of this prototype was, however, very disappointing -- it was capable of supporting only a very small number of (very patient) users. Given this experience, we recently embarked on a reimplementation of the system using the file system-based design described above and programmed in C and C++. A prototype usable for developing browsers and other applications will be completed by August, 1992. We plan to have a version suitable for distribution to other groups by mid-year 1993.

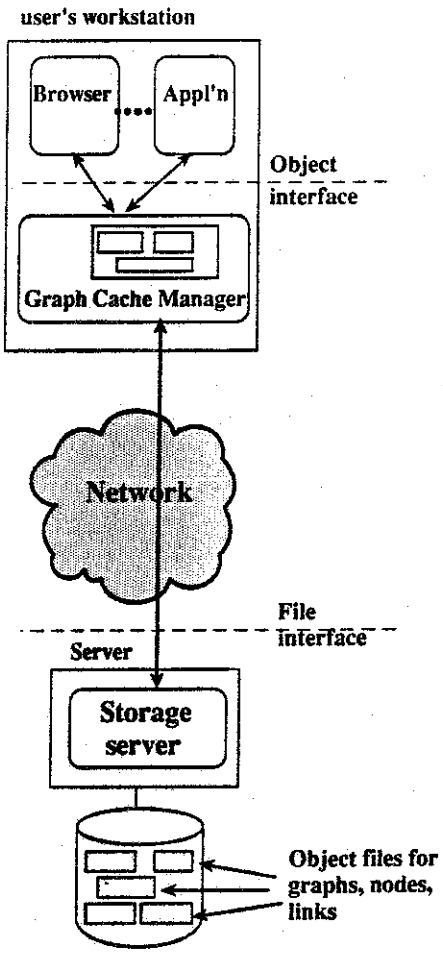

Figure 6: Basic System Structure

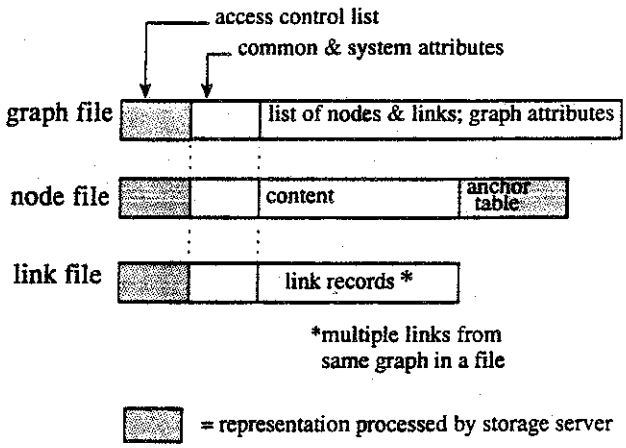

Figure 7: Object File Content

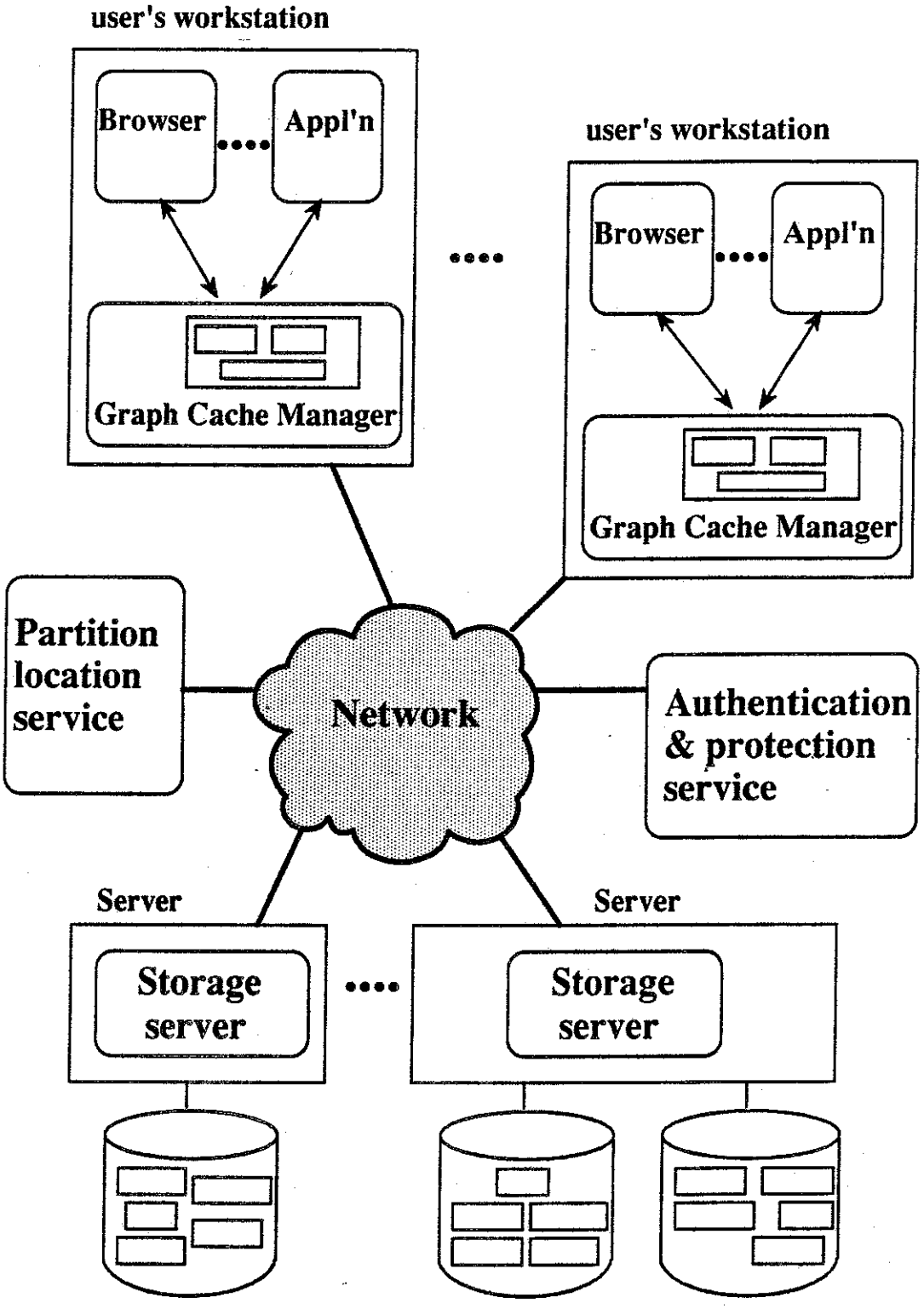

Figure 8: Complete System Structure

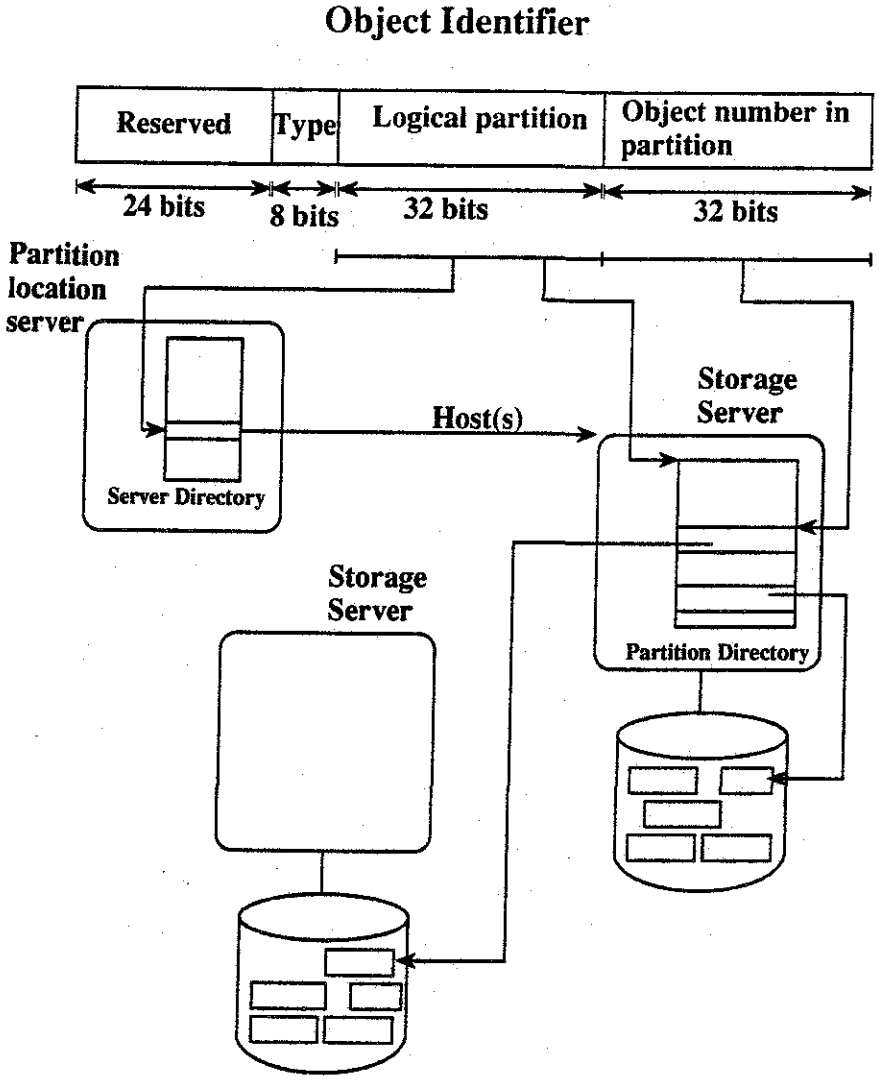

Figure 9: OID and Object Location

7. Review of Related Work

Although many systems provide some support for groups, no system currently supports collaborations among large (10-100 persons), widely distributed groups. Current systems differ widely on factors such as the data model supported, scalability, concurrent reader/writer semantics, and protection. Because hypermedia systems and applications provide many functions required in our storage system, we will briefly compare ABC/DGS with several hypermedia systems that have significant capability for supporting collaborating groups (Intermedia: [Haan, 92], Yankelovich, 1988]; Telesophy: [Caplinger, 1987], [Schatz, 1987]; HyperBase: [Schutt & Streitz, 1990]; KMS: [Akscyn, 1988]; Augment: [Engelbart, 1984]; and HAM: [Campbell & Goodman, 1988], [Delisle & Schwartz, 1986], [Delisle & Schwartz, 1986]).

A distinguishing characteristic of a hypermedia system is its data model. While most systems include some concept of a collection of objects and a notion of object-to-object reference links, each system has its own flavor. For example, Intermedia and DGS provide explicit mechanisms for associating links with anchor points within nodes, whereas HyperBase and HAM suggest that applications use attributes to store this information. An advantage of the former approach is that it allows the storage system to provide guarantees about the consistency of link and anchor information (e.g., eliminating "dangling" anchors).

Table 2 characterizes each of the systems listed above, based on features such as whether nodes and links can be collected into named groups (called aggregates), whether these aggregates can themselves be grouped, and a description of which object types can be the endpoints of links. We also include the Dexter Reference Model (an abstract description of a generic hypermedia data model) [Halasz, 1990] in Table 2 even though there is no existing system implementation.

Support for a full spectrum of aggregate types is a distinguishing feature of DGS, HyperBase, and the Dexter model. However, in contrast to the last two models, our data model disallows links that have aggregates or other links as their endpoints. Whereas Shutt and Streitz [1990] advocate links to links, we are concerned that this complicates the data model, making it more difficult for humans and automated agents to maintain the consistency and completeness of artifacts over time. For similar reasons, our data model requires that a link to an aggregate be represented using an equivalent but simpler construct, i.e., by creating a link to the *node* which contains the aggregate. As a result, DGS has the full power of aggregates without sacrificing the elegance of its graph theoretic data model.

Hypermedia System	Aggregate Object	Aggregates of		Nodes can be in more than one aggregate?	Links can be in more than one aggregate ?	Aggregates of Aggregates	Endpoints of links	Strongly Typed Aggre-gates
		Nodes	Links					
Intermedia	Webs	Yes	Yes	Yes	No	No	Nodes	No
Telesophy	Composite Information Units	Yes	No	Yes	No	Yes	Nodes, Aggregates	No
HyperBase	Complex Objects	Yes	Yes	Yes	Yes	Yes	Nodes, Links, and Aggregates	No
Dexter	Composite Components	Yes	Yes	Yes	Yes	Yes	Nodes, Links, and Aggregates	No
Augment	Hierarchically Organized Files	Yes	No	No	No	No	Nodes	No
HAM	Contexts	Yes	Yes	No[1]	No	No	Nodes	No
DGS	Graphs	Yes	Yes	Yes	Yes	Yes	Nodes and Aggregates[2]	Yes

1 Can be overcome by using cross-context links

2 Link to node containing agggregate

Table 2: Data Model Features of Selected Hypermedia Systems

Another area in which systems differ substantially is in the reader/writer semantics and protection mechanisms that they provide (see Table 3). Many systems have recognized the need for flexible protection mechanisms. However, DGS is the only system that provides an *administrate* permission that allows users to assign capability to change access authorizations to others. This feature is especially important since the lifetime of the next generation of hypermedia data may span decades [Malcolm, et. al., 1991]. Thus, responsibility for protecting an object may change hands many times.

Hypermedia System	Concurrent Reader/Writer Semantics	Protection of Objects
Intermedia	Supports multiple users reading and annotating, and a single writer. First user to write an object locks out other potential writers.	Provides read, write, and annotate permissions that can be granted to users and groups of users.
Telesophy	Supports multiple concurrent readers and writers. When writers overlap, the last writer completely overwrites the work of the others.	could not be determined
HyperBase	could not be determined	could not be determined
KMS	Uses an optimistic concurrency method. When a writer attempts to save a node, he may be denied because someone else has concurrently written to the same node. In this case, the human user must manually merge the two conflicting versions.	Owner can protect a frame from modification or read access. In addition, an intermediate form allows users to add annotation items, but not to modify existing items.
Augment	Can have multiple readers of documents that have been submitted to the Journal system.	Objects in the Journal are read-only. Access to Journal entries can be restricted at submission time.
HAM	could not be determined	Access Control Lists (optional): Access, annotate, update, and destroy permissions.
DGS	Supports multiple non-annotating readers and a single writer OR multiple annotating and non-annotating readers. Applications must declare their intent at the time that they open an object. Intent can be one of: read and annotate; read only; read/write and annotate.	Access Control Lists: Access (read or read/write) and administrate permissions. Rather than associate a single annotate permission with a node, the DGS provides a more flexible mechanism of associating annotate permission with the graphs which contain the node. Thus, a user might be allowed to annotate a node within his personal context at the same time that he is denied the ability to annotate the node in a public context.

Table 3: Concurrent Reader/Writer Semantics and Object Protection

These systems also differ in their capability to scale up to large numbers of users (and objects) while preserving the illusion of location transparency. Both Telesophy and DGS have made scalability a central issue in their designs. However, DGS provides more flexibility in its data model and stronger consistency semantics. In addition to Telesophy and DGS, the Distributed Hypertext approach of Noll and Scacchi [1991] is also noteworthy since they use hypertext to integrate diverse information repositories that are distributed across a wide-area network.

Some additional dimensions one could use to compare hypermedia systems are support for an open architecture, change notification, and versioning. A system is considered *open* if it allows non-hypermedia applications to be integrated easily (as DGS does). The Sun Link Service [Pearl, 1989] is also a noteworthy example of an open hypermedia system. We say that a system supports notification if it provides methods for users and applications to be notified of actions that are taking place in the storage system. For example, a user may wish to be notified when a particular node is updated. Despite the importance of notification for distributed, asynchronous collaboration, no current system (including DGS) appears to support this well, although some promising work on this problem is taking place at Purdue [Dewan, 1991]. Finally, HAM's use of contexts [Delisle & Schwartz, 1987] to provide versioning and its notification support are still noteworthy and unique.

8. Summary and Conclusions

Collaborative groups face many problems, but one of the hardest and most important is to meld their thinking into a conceptual structure that has integrity as a whole and that is coherent, consistent, and correct. Seeing that construct as a single, integrated artifact can help. But groups must also be able to view specific parts of the artifact in order to understand it and to manage it. Our Distributed Graph Storage System is guided by these requirements, along with others discussed above. The graph-based data model permits us to both partition the artifact and to compose those pieces to build larger components and the whole. The distributed architecture, in turn, permits us to build a system that can scale up in terms of the size of the artifact, the number of users, and their geographic distances from one-another.

As we look to the future, additional issues we will explore pertain to wide-area network access, content search, notification, graph traversal, and support of a richer set of graph and set operations and queries.

9. Acknowledgments

A number of individuals and organizations have contributed to this project. Gordon Ferguson and Barry Elledge contributed to the Smalltalk prototype that preceded DGS. Mike Wagner, Zhenzin Wang and Shankar Khrishnan have contributed to the implementation of DGS. This work was supported by the National Science Foundation (Grant # IRI-9015443) and by the IBM Corporation.

10. References

Akscyn, R.M.; McCracken, D.L.; & Yoder, E.A. (1988). KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations, *Communcations of the ACM*, 31(7), July 1988, 820-835.

Baker, M. G.; Hartman, , J. H.; Kupfer, M. D. ; Shirriff, K. W. ; & Ousterhout, J. K. (1991), Measurements of a Distributed File System, *Operating Systems Review*, 25(5), Special Issue: *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, October, 1991, pp. 198-212.

Brooks, F. P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), 10-19.

Campbell, B.; & Goodman, J.M. (1988). HAM: A General Purpose Hypertext Abstract Machine, *Communications of the ACM*, 31(7), 856-861.

Caplinger, M. (1987). An Information System Based on Distributed Objects. *OOPSLA 87 Proceedings*, October 4-8, 1987.

Delisle, N.M.; & Schwartz, M.D. (1986). Neptune: A Hypertext System for CAD Applications, *SIGMod Record*, 15(2), June 1986, 132-142.

Delisle, N.M.; & Schwartz, M.D. (1987). Contexts-A Partitioning Concept for Hypertext, ACM *Trans. on Office Information Systems*, 5(2), pp. 168-186.

Dewan, P. (1991). Flexible Coordination in Collaborative Software Engineering, Coordination Theory and Collaboration Technology Workshop, National Science Foundation, Washington, D.C., June 1991, pp. 41-48.

Engelbart, D.C. (1984). Authorship Provisions in AUGMENT, COMPCON '84 Digest, *Proceedings of the 1984 COMPCON Conference*, San Francisco, CA, Feb 27- Mar 1, 1984, pp. 465-472.

Haan, B.J., et. al. (1992). IRIS Hypermedia Services, *Communications of the ACM*, 35(1), January, 1992, .

Halasz, F. (1987). Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Proceedings of Hypertext '87*, pp. 345-365.

Halasz, F.; & Schwartz, M. (1990). The Dexter hypertext reference model, *Proceedings of the Hypertext Standardization Workshop* (Gaithersburg, Maryland), 1-39.

Howard, J. H.; Kazar, M. L.; Menees, S. G.; Nichols, D. A..; Satyanarayanan, M.; Sidebotham, R. N.; &West, M. J. (1988). Scale and Performance in a Distributed File System, *ACM Transactions on Computer Systems*, 6(1), February 1988, 51-81.

Malcolm, K.C.; Poltrock, S.E.; & Douglas, S. (1991). Industrial Strength Hypermedia: Requirements for a Large Engineering Enterprise, *Proc. Hypertext 91*, ACM, New York, 1991, pp. 13-24.

Nelson, M.N.;Welch, B.B. ; & Ousterhout, J. K. (1988). Caching in the Sprite Network File System, *ACM Transactions on Computer Systems*, 6(1), February 1988, 134-154.

Noll, J.; & Scacchi, W. (1991). Integrating Diverse Information Repositories: A Distributed Hypertext Approach,*Computer*, 24(12), December 1991, 38-45.

Pearl, A. (1989). Sun's Link Service: A Protocol for Open Linking, *Proc. Hypertext 89*, ACM, New York, 1989, pp. 137-146.

Schatz, B.R. (1987). Telesophy: A System for Manipulating the Knowledge of a Community, *Proc. Globecom 87*, ACM, New York, 1987, pp. 1181-1186.

Schutt, H.; & Steitz, N. (1990). HyperBase: A Hypermedia Engine Based on a Relational Database Management System, Integrated Publication and Information Systems Institute, West Germany, submitted to ECHT '90.

Smith, J.B.; & Smith, F. D. (1991). ABC: A Hypermedia System for Artifact-Based Collaboration. *Proceedings of Hypertext '91*, San Antonio, TX, Dec. 1991, pp. 179-192.

Yankelovich, N., *et. al.* (1988). Intermedia: The Concept and the Construction of a Seamless Information Environment, *Computer*, 21(1), January 1988, 81-96.