A Distributed Data-Storage Service for Supporting Group Collaborations

TR92-044

Douglas E. Shackelford
John B. Smith
F. Don 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 supported by the National Science Foundation (Grant #IRI-9015443 and by IBM Corporation (SUR Agreement #866).

A Distributed Data-Storage Service for Supporting Group Collaborations

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

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

Corresponding author:
John B Smith
jbs@cs.unc.edu
(919) 962-1792

Abstract: The architecture and implementation of a distributed data-storage service is presented. Requirements for this design were determined from our research on computer and communications systems to support groups of users collaborating to solve complex problems such as the design of large software systems. The data model supported by the system is much richer than that of conventional file systems and is capable of directly supporting hypertext applications such as (interactive) navigational browsers. Some important issues addressed in this design include scale, performance, concurrency semantics, access protection, location independence, and replication (for fault-tolerance). We show that even with the richer functions supported, a high level of performance and scalability can be achieved. The key to this result is our application of design principles (e.g., distribution of computationally-intensive work, caching, minimal client-server interaction) found in some conventional distributed file systems. The design also avoids the necessity for more complex mechanisms such as distributed transaction processing.

Keywords: distributed data, data models, computer-supported cooperative work (CSCW), distributed file systems, performance, hypertext.

A Distributed Data-Storage Service for Supporting Group Collaborations

1. Introduction and motivation

Our research focuses on the process of collaboration and on technology to support that process. We are concerned with the intellectual collaboration that is required for designing software systems or 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 is building the "conceptual construct." To facilitate communication among team members, groups generate a variety of interrelated elements of their artifact. Elements of the artifact may also become part of the final product. For software systems, the artifact may include concept papers, architecture, requirement, or specification documents, 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. A subtle but important point is that we view a group's tangible creations as parts of a single artifact. 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, 1991] to support that process.

In the following sections we discuss the distributed data-storage service for group artifacts being developed as part of ABC. We first describe briefly in section 2 the underlying data model and introduce concepts necessary to understand design decisions presented in the following sections. Section 3 gives key requirements for the distributed implementation; section 4 discusses concurrency semantics and access protection; section 5 sketches the distributed implementation, some early performance results and the current status; section 6 relates our design to other work, and section 7 gives a summary.

2. Data Model

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 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. Such relationships, if made explicit, can greatly enhance the group's ability to locate information and to maintain the coherence, completeness, and correctness of the materials. For this reason, we believe that hypertext plays a key role in computer systems for supporting collaboration and we have designed a data model that supports hypertext applications such as interactive browsers and other navigation aids [Haan, 1992].

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. The data model explicitly encourages users to organize information according to principles of modularity and decomposition.

The most basic element of artifact storage is a node. Nodes are repositories for information stored either as node attributes or as node content. Node attributes are named variables with few restrictions on their type and size. Some attributes (such as creation time and size) are maintained automatically by the system. There may also be a number of application-defined and maintained attributes on any node. Node content is used in two ways -- for data storage it can contain any data represented as a stream of bytes (the familiar model used in conventional file systems); a second use is described in the next paragraph. 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-content in our system with no changes. The ABC system includes a mechanism for invoking an arbitrary application and providing it with input from node content that appears to the application as an ordinary file.

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, a node's content 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 or as link data-content.

In addition to the basic S-graph, the data model also provides a predefined set of strongly typed S-graphs. Currently five types are supported: 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 use of these rather abstract mechanisms is best illustrated by examples. The data model encourages users to decompose a large artifact into small S-graphs related by composition. This organization improves human comprehension and increases potential for concurrent access to individual components. Figure 1 illustrates a document structure and Figure 2 shows a more complex example -- organizing public and private parts of an artifact (node content is indicated by dashed lines). Note that the example shown in Figure 2 subsumes the organization of data in a conventional file system (consider S-graphs as directories and nodes with data content as files) while adding new functions for representing structure among nodes within an S-graph (directory).

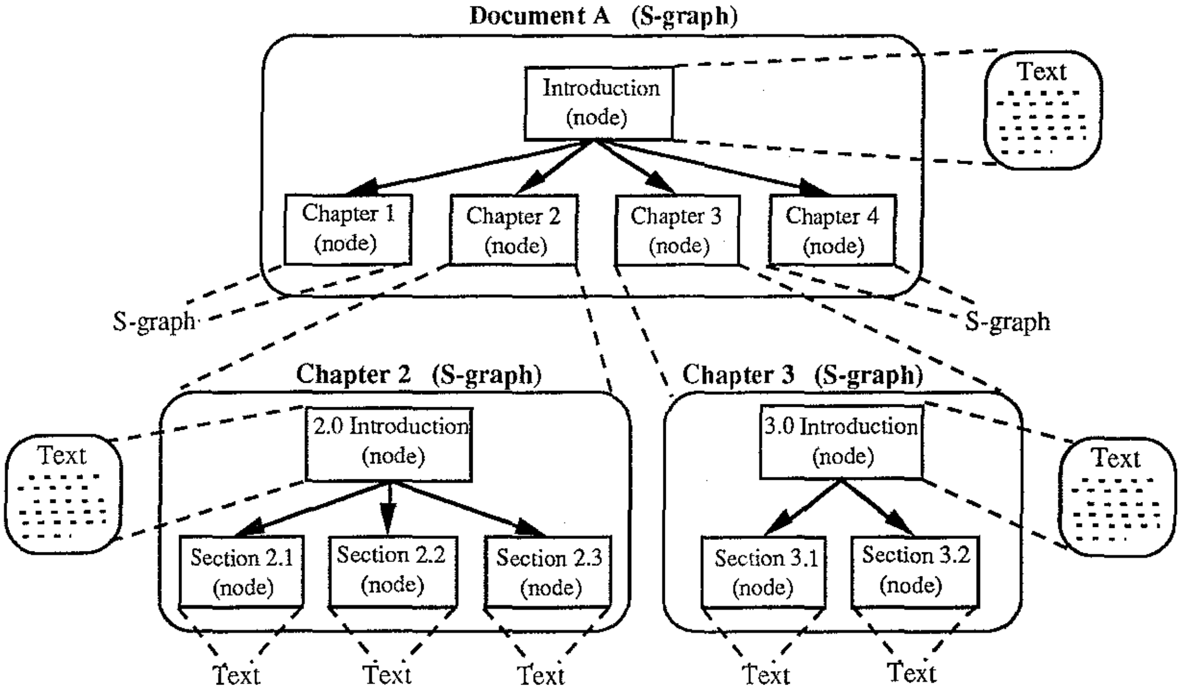

Figure 1: Using Node Content and Structure to Represent a Document

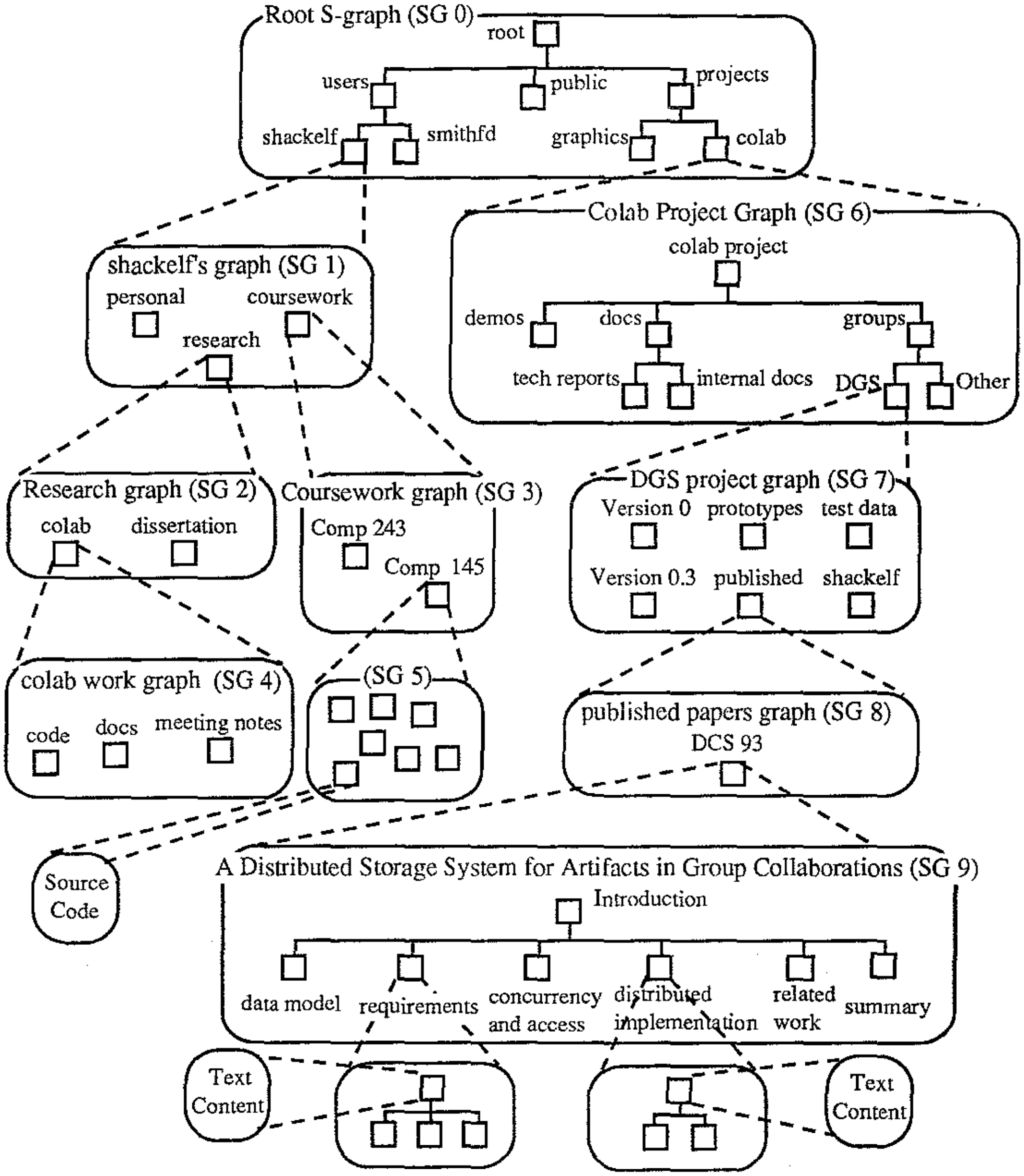

Figure 2: Organizing Public and Private Parts of an Artifact

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 3). H-links and the nodes they link are grouped into hypergraphs (H-graphs). Links similar in function to H-links are usually the key elements of conventional hypertext systems.

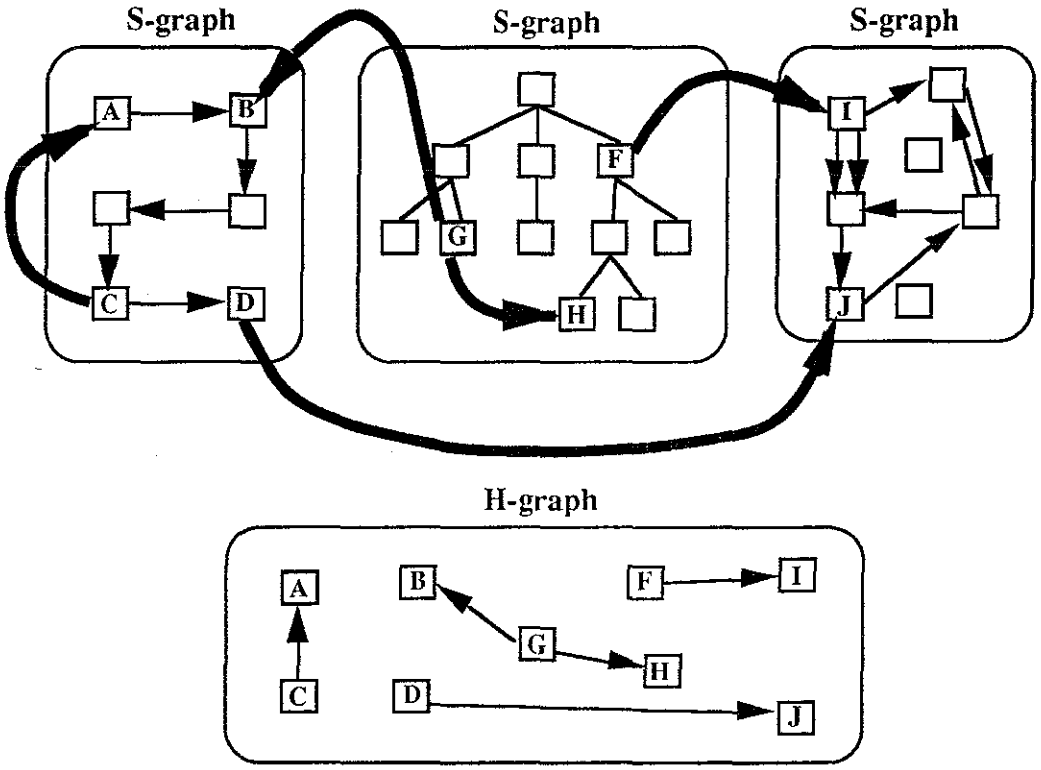

Figure 3: H-Links and H-Graph

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 the text of a document node to its definition in the text of a glossary node. 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 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 4).

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.

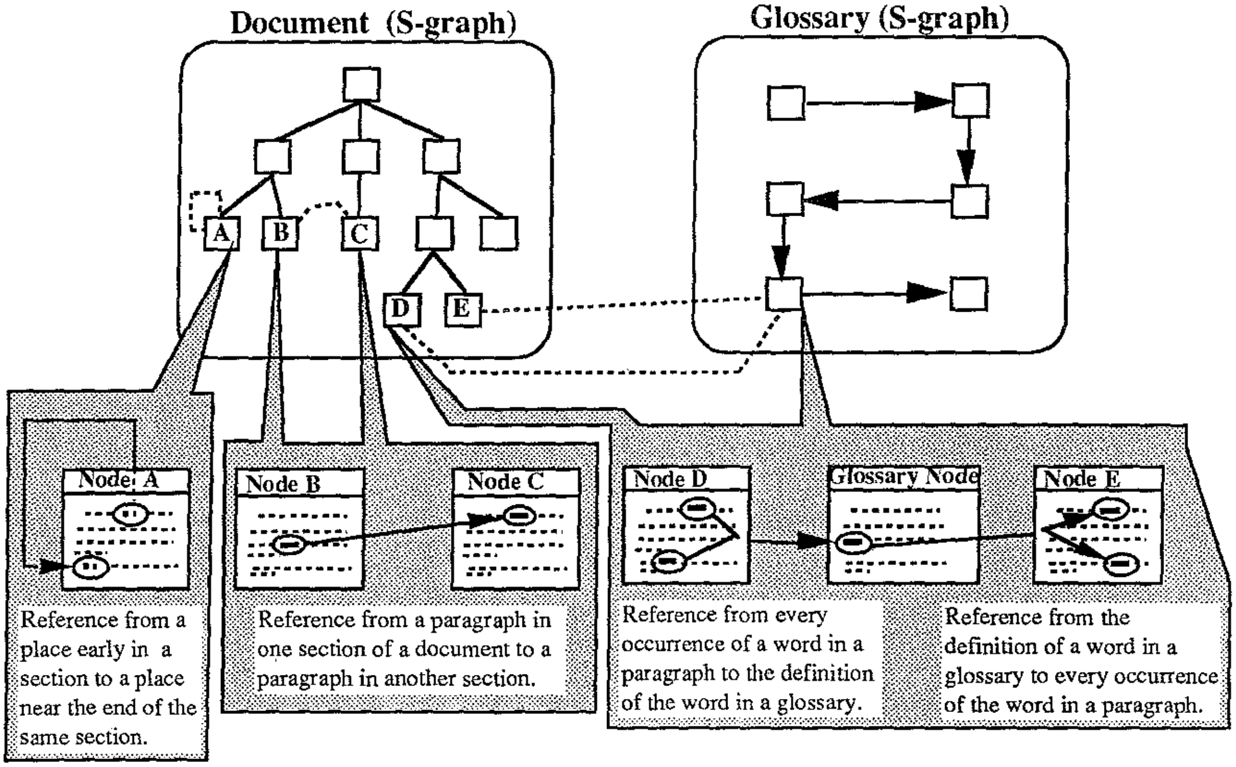
Document (S-graph)
Glossary (S-graph)
A
B
C
D
E
Node A
Node B
Node C
Node D
Glossary Node
Node E
Reference from a place early in a section to a place near the end of the same section.
Reference from a paragraph in one section of a document to a paragraph in another section.
Reference from every occurrence of a word in a paragraph to the definition of the word in a glossary.
Reference from the definition of a word in a glossary to every occurrence of the word in a paragraph.

Figure 4: Examples of anchored hyperlinks

Some attributes of 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 -- attributes that are meaningful only in the context of one graph that contains the node or link (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 (X,Y) screen position of its drawing of the node in a graph attribute for the tree S-graph, while a list browser would store its (X,Y) position of the node in a graph attribute for the list S-graph. Graph attributes of a node store the information most often used by interactive browsers (because the visual presentation of nodes in a graph is inherently context-dependent).

3. Requirements for the Distributed Storage Service

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

Permanent (persistent) storage -- obvious but fundamental.

Sharing with protection -- because the artifact effectively constitutes the group's collective memory, it must be sharable by all. There are, however, requirements for mechanisms to authorize or deny access to selected elements of the artifact by individuals or sub-groups.

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. To be scalable, it must be possible to distribute the system over available processing and network resources and to add resources incrementally as necessary. Performance (responsiveness) as perceived by users must not degrade significantly as the system grows in scale.

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 is 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 must 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.

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

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 on node data-content with no changes.

4. Concurrency and Access Protection for Groups

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 write them (in the following discussion, we refer to a user's application that changes an object as a "writer" and one that only views an object as a "reader"). Given strong support for an artifact composed of small elements, we expect multiple concurrent writers of an object to be quite rare (studies of distributed file systems [Baker, 1991], [Kistler, 1991] show this to be true even when used for group projects). Multiple concurrent readers of an object will, however, be more 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 creating anchored H-links, the node must be open in read mode and the user must have read_write permission on the link that is being anchored. 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 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. Browsers and other applications must implement measures for deadlock avoidance in case an access request is not 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.

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. 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.

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. Distributed Implementation

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

Some examples are:

Access all the links from node 6 in H-graph 50.
Access the value of attribute "XY position" of node 10 in S-graph 100.
Access all the sibling nodes of node 25 in S-graph 100.

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

In section 2 we showed how the data model encourages and supports composition of large artifacts from smaller components. 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 the people who most need it. This is especially important when groups are widely distributed geographically.

Given an artifact composed from small elements and user access via interactive 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, 1991], [Kistler, 1991]. Our design is based on the notion that a scalable implementation can be achieved by applying design principles such as local caching, bulk-data transfer, and minimal client-server interactions pioneered in high-performance, scalable file systems like Andrew [Howard, 1988], Sprite [Nelson, 1988], and Coda [Kistler, 1991]. We also model our approaches to data consistency, concurrency semantics, and replication after these distributed file systems. This provides a sufficient level of function to users without requiring the full complexity of mechanisms (e.g. distributed transactions) used in database systems.

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.

The basic structure of the system is shown in Figure 5. A browser or application process acts on behalf of a user to read and modify objects. Each user's workstation runs a single graph-cache manager (GCM) process that services all applications running on that machine. Application requests are directed over local interprocess communication facilities to the GCM. The GCM maintains a local copy of node, link, and graph objects used by application processes and is responsible for implementing all operations on objects in the data model except for anchor table merging. The GCM is also responsible for maintaining the consistency of typed S-graphs. It is important to note that this design distributes the processing for all complex object operations to the users' workstations and thus minimizes the processing demands on shared (server) resources.

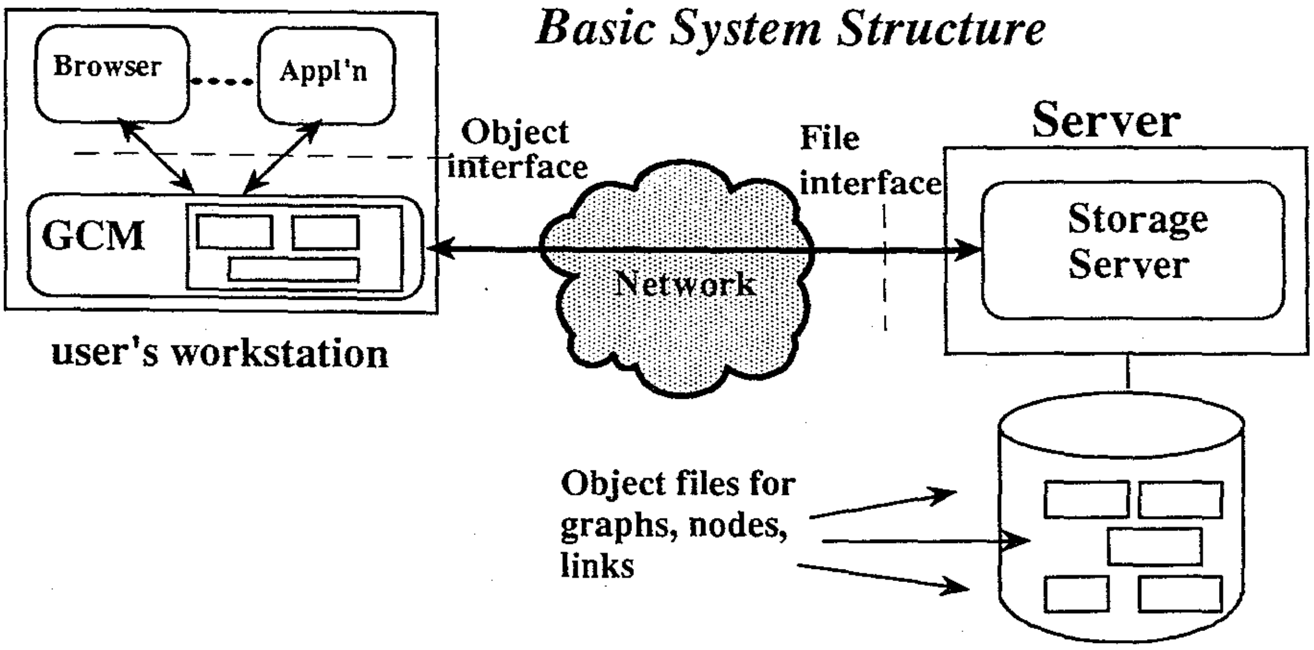

Figure 5: Basic system structure

When an application opens an object, the GCM, in turn, opens the object at the storage server and retrieves it using a whole-file transfer. The received object is converted from its representation in a file to an object representation designed for fast access in memory. As the application makes requests, the GCM performs those operations on the copy in its local cache. Write operations are reflected in the storage server only when the GCM closes the object and returns the modified file representation to the storage server. Each file retrieved from the storage server contains either a whole node (including data content, if present), a whole graph, or a group of links. An important performance optimization is that context-dependent attributes (graph attributes) and S-link information for all nodes in a graph are stored in the graph file. Thus, all of the data needed by a browser to display a graph is available from a single request (open) to the storage server. The structure of each type of file is shown in Figure 6. Nodes and graphs are stored individually, whereas links are grouped according to the graph in which they were created.

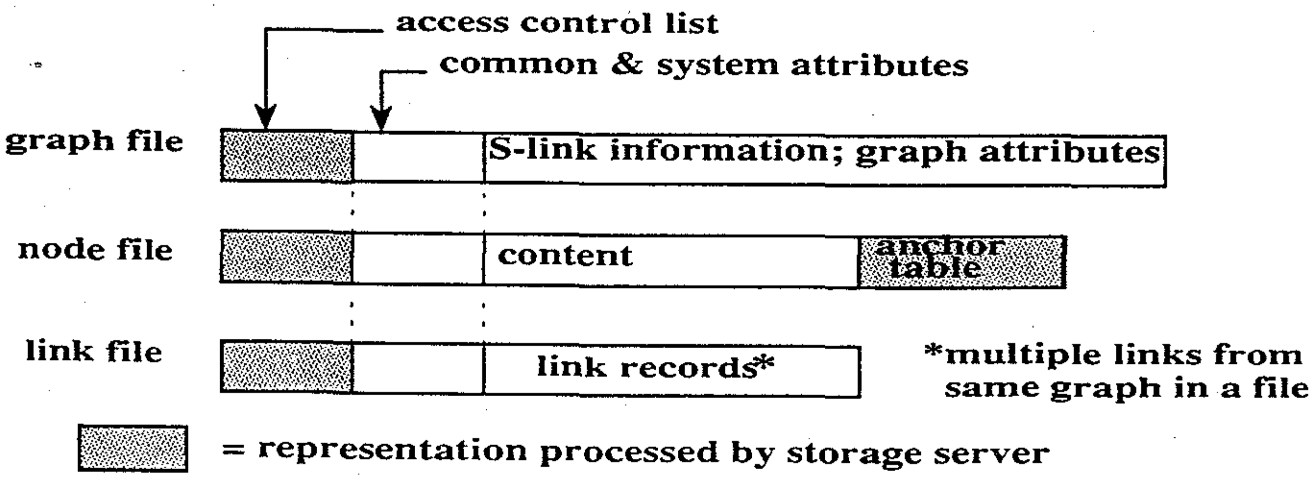
access control list
common & system attributes
graph file
S-link information; graph attributes
node file
content
anchor table
link file
link records*
*multiple links from same graph in a file
= representation processed by storage server

Figure 6: Structure of object files

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 are also responsible for maintaining access control lists, enforcing access authorizations, enforcing concurrency semantics, creating unique OIDs and anchor IDs, and 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 for the same object. An open request will fail if the user lacks proper access authorization or if the open conflicts with other opens in progress.

Each GCM may need to communicate with multiple storage servers. Figure 7 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. Each partition is associated with real storage devices. 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 (logically) remain in the same partition for its entire lifetime because its OID cannot be changed.

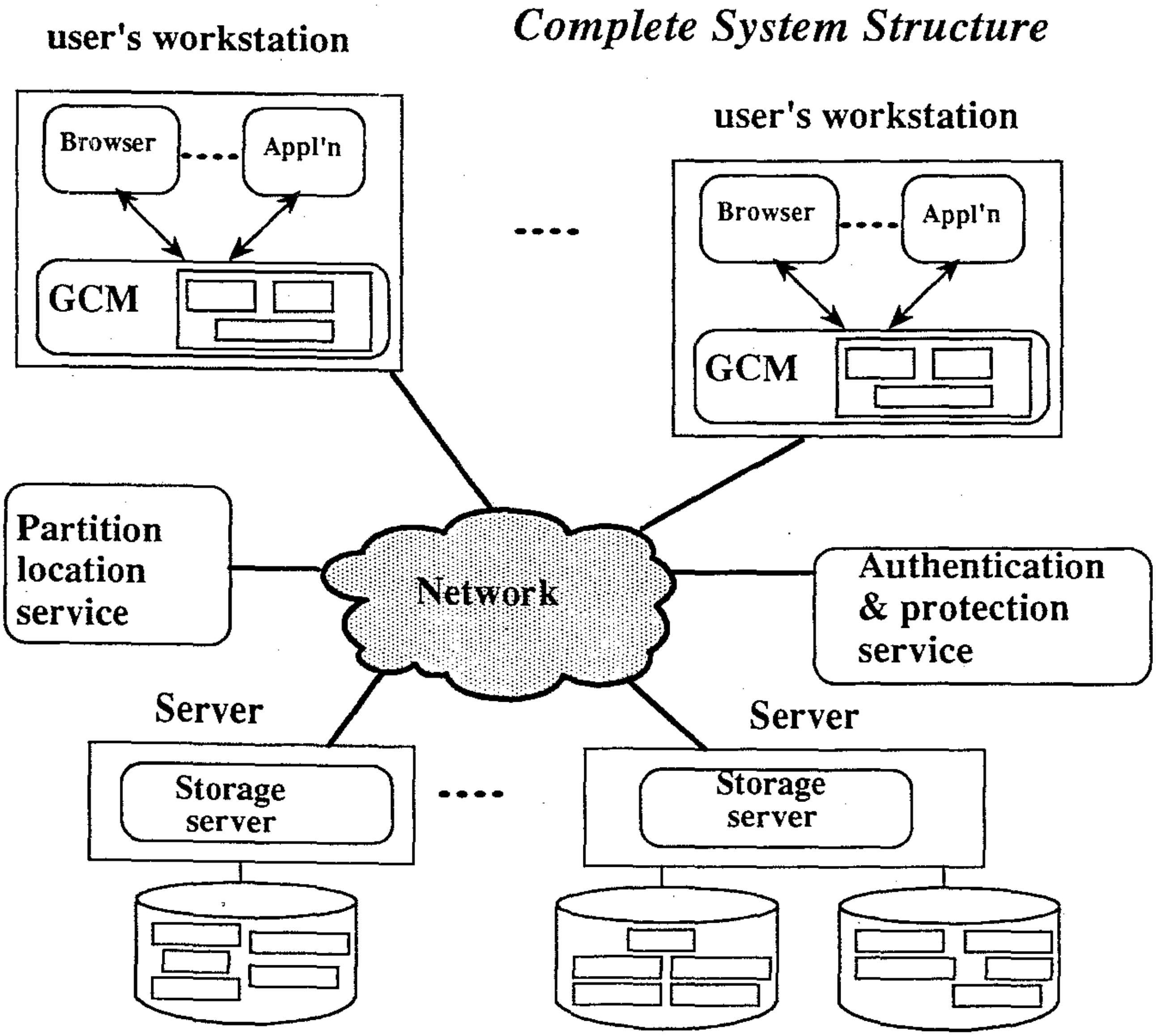

Figure 7: Complete system structure

We distinguish the partition number of an object from its absolute physical location(s) and, by introducing a level of indirection (a partition directory), it is possible to change the physical location of an object while preserving its OID and, therefore, all its link and composition relationships with other objects (see Figure 8). Partition-location servers maintain a mapping of logical partitions to host(s) running server processes for that partition. The GCM extracts the partition number from the OID of the object and uses the partition location service to find the host running a storage server process maintaining a directory for that partition (the GCM can also cache the partition location information for use in references to other objects). We expect that in most cases one storage server maintains both the partition directory and data storage for an object. Despite their importance, partitions are invisible to users. Only system administrators and system programmers need to understand partitions. An RPC interface to the storage servers is provided for administrative processes to use in creating new partitions, moving objects from one physical partition to another, and performing backup and recovery operations.

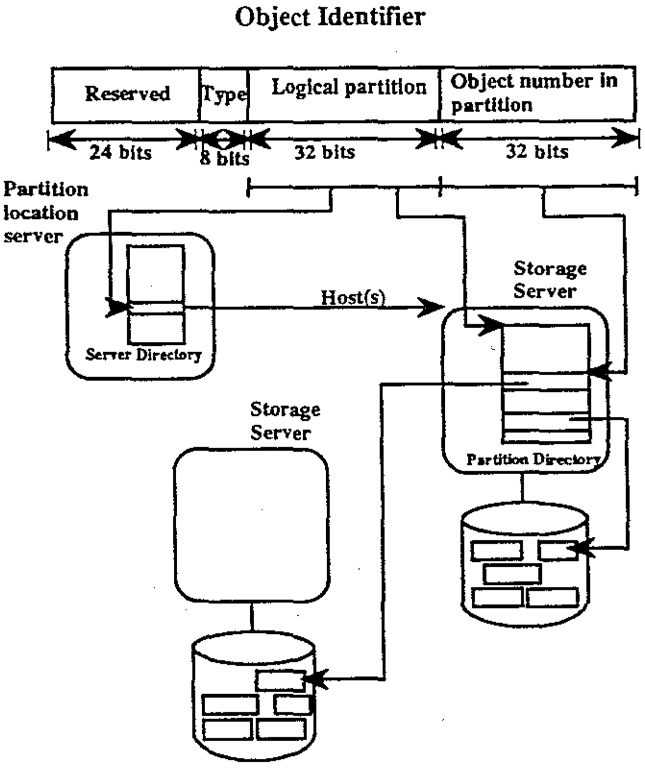

Figure 8: OID and object location

Storage servers are responsible for managing partitions on disk, replicating partitions for availability and fault tolerance in case of media or process failures, and for recovering from most failures. The key to our implementation of fault tolerance is the ISIS system developed by Ken Birman and his colleagues at Cornell University [Joseph, 1986]. In particular, we use ISIS process groups to maintain replicated copies of physical partitions and to provide the location independence of logical partitions. Each logical partition corresponds to an ISIS process group.

Performance and scalability are two key requirements for the system (see section 3). To evaluate the current implementation with respect to these requirements, we have begun a series of benchmark experiments similar to those used to evaluate performance and scalability of distributed file systems such as Andrew [Howard, 1988] and Sprite [Nelson, 1988]. We have created several benchmark programs designed to stress different aspects of the system. The most interesting of these is a "synthetic browser" program that mimics the requests that result when users search for information in an artifact stored in the system. Load on the storage service is generated by running copies of the program on several workstations. This program has parameters that can be used to produce a wide range of browsing behaviors. In our first experiments we used parameter values that represent the behavior of human subjects we observed in a series of experiments conducted to understand how people would use a hypertext system for problem solving [Smith, 1992]. With these values, each instance of the program running on one workstation generates a load on the server corresponding to approximately 10 users working with interactive browsing applications. We have also written an "artifact generator" program that, based on a number of input parameters, creates a structure of graphs, nodes, and links to serve as data for the browsing benchmark.

The results of our initial measurements have been very encouraging. The configuration for these measurements consisted of one storage server running on a DECstation 5000/25c and up to 7 workstations (DECstation 5000/120s) each running a copy of the benchmark program. All processors are interconnected by a single ethernet segment. Load on the server for each run was determined by the number of workstations executing the synthetic browser. The most significant results are:

- CPU utilization on the server is at most 0.5-1.0% per active user.
- Server response times to requests from 50 users increased by less than 20% over response times to requests from 10 users.

- At loads that would be generated by 60-70 users, the local disk on the server appears to become the limiting resource.

These results show that one server workstation can support at least 50 users (especially if it is equipped with more suitable disks). We believe this result is an important indication that the service will be scalable to support large groups. More extensive benchmark experiments are underway to validate this conclusion for a variety of configurations.

An initial prototype of the storage system using a different design (essentially a distributed database) was implemented in Smalltalk over a year ago. Experiences from constructing and using that prototype were invaluable in refining the requirements, data model, and programming interface. The performance of that prototype was, however, very disappointing -- it was capable of supporting only a very small number of (very patient) users. Given this experience, we recently completed a redesign and reimplementation of the system based on a distributed file system design (instead of a database-like design) and programmed in C and C++. The performance of this version is quite good (see discussion above) and is now being used for developing browsers and other collaboration support tools. We continue to make enhancements (mostly for operations and administration) and plan to have a version suitable for distribution to other groups by mid-year 1993.

6. Review of Related Work

Because hypertext systems and applications provide many of the functions required in our storage system, we will briefly compare our design with several hypertext 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, 1987]). We also include the Dexter Reference Model (an abstract description of a generic hypertext data model) [Halasz, 1990] even though there is no system implementation. These systems differ widely on factors such as the data model supported, scalability, concurrent reader/writer semantics, and protection. In the following we use "ABC" to refer to the storage service for our ABC system.

Support for a rich data model (including aggregation (named groups), composition of aggregates, and aggregates as end points of links) is a distinguishing feature of ABC, HyperBase, and the Dexter model.

Intermedia, HAM, and Augment do not use aggregates in composition or linking. Telesopy has these functions but excludes links from aggregates. Only ABC provides strongly typed aggregate objects.

Other areas in which these systems differ substantially are in the semantics of concurrent reading and writing and in the access protection mechanisms (see Table 1). 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 ABC have made scalability a central issue in their designs. However, ABC provides more flexibility in its data model and stronger consistency semantics. In addition to Telesophy and ABC, 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. A system is considered open if it allows non-hypertext applications to be integrated easily (as ABC does). The Sun Link Service [Pearl, 1989] is also a noteworthy example of an open hypertext system.

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.
ABC	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 ABC 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 1: Concurrent Reader/Writer Semantics and Object Protection

7. Summary

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 design was 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.

8. Acknowledgments

A number of individuals and organizations have contributed to this project. Gordon Ferguson and Barry Elledge helped develop the Smalltalk prototype. Xiaofan Lu, Rajaraman Krishnan, Shankar Khrishnan, Mike Wagner, and Zhenxin Wang have contributed to the current implementation. This work was supported by the National Science Foundation (Grant # IRI-9015443) and by the IBM Corporation.

9. 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.

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.

Joseph, T. A.; & Birman, K. P. (1986). Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems, *ACM Transactions on Computer Systems*, 4(1), February 1986, pp. 54-70.

Kistler, J. J.; & Satyanarayanan, M. (1991) Disconnected Operation in the Coda 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. 213-225.

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, D.K. (1992). Hypermedia vs. Paper: User Strategies in Browsing SNA Materials, *Technical Report TR92-036*, Department of Computer Science, University of North Carolina at Chapel Hill.

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.