The Architecture and Implementation of a Distributed Hypermedia Storage System

Douglas E. Shackelford
Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina
27599-3175
shackelf@cs.unc.edu

John B. Smith
jbs@cs.unc.edu

F. Donelson Smith
smithfd@cs.unc.edu

ABSTRACT

Our project is studying the process by which groups of individuals work together to build large, complex structures of ideas and is developing a distributed hypermedia collaboration environment (called ABC) to support that process. This paper focuses on the architecture and implementation of the Distributed Graph Storage (DGS) component of ABC. The DGS supports a graph-based data model, conservatively extended to meet hypermedia requirements. Some important issues addressed in the system include scale, performance, concurrency semantics, access protection, location independence, and replication (for fault tolerance).

KEYWORDS

distributed data, computer-supported cooperative work (CSCW), distributed file systems, performance, scalability, hypertext

INTRODUCTION AND MOTIVATION

Future hypermedia systems will integrate diverse information resources, systems, and technologies. They will be based on modular architectures (e.g., (Thompson, 1990)) that separate orthogonal concerns into plug-compatible components such as change management, query and content search, notification, application-specific concurrency control, computational semantics, and window conferencing. Some of these components, such as change management, may be highly dependent on the semantics of a particular domain, whereas others will provide general support for all applications.

> The key point ... is that it is modular and open. This modularity is based on the observations that the functions the modules perform are independent of each other, that is *orthogonality implies modularity.* (Thompson, 1990)

Orthogonality implies modularity; *modularity implies choice.* The importance of this observation is that every service has a cost associated with it. For example, transactions may be the appropriate concurrency mechanism for one application, while imposing prohibitively high overhead on another. Ideally, one should be able to use a service when it is needed without having to pay for it when it is not.

In this paper, we describe the architecture and implementation of our Distributed Graph Storage (DGS) system. We have designed it in a way that supports modular expansion to add services such as those enumerated above. A fundamental requirement has been that the basic hypermedia services for data storage and access should be inexpensive, efficient, and scalable. This is particularly important since the performance of these basic services is an upper bound on the performance of the system as a whole.

The DGS has been developed as a part of a larger program of research that 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. For software systems, the artifact may include concept papers, architecture, or specification documents, programs, diagrams, reference and user manuals, as well as administrative documents. 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 and Smith, 1991) to support that process. ABC has six key components(Jeffay et al., 1992): the Distributed Graph Storage system, a set of graph browsers, a set of data application programs, a shared window conferencing facility, real-time video and audio, and a set of protocol tools for studying group behaviors and strategies.

In the following sections we discuss the architecture and implementation of the DGS. We first describe briefly in Section 2 the key requirements for the distributed implementation. Section 3 presents the underlying data model. Section 4 sketches the distributed implementation, our efforts to evaluate the system's performance, and the current status. Section 5 relates our design to other work, and we conclude in Section 6.

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.

DATA MODEL CONCEPTS

ATTRIBUTES AND CONTENT

The most basic element of the data model is the *node*, which usually contains the expression of a single thought or idea. Structural and semantic relationships between nodes are represented explicitly as *links* between nodes.[1]

The data model provides two mechanisms for storing information within a node: node attributes and node content. *Attributes* are typed, named variables for storing fine-grained information (approximately 1-100 bytes). Some attributes (such as creation time and size) are maintained automatically by the system. There may also be an unlimited number of application-defined attributes.

In comparison to attributes, node *content* is designed to reference larger amounts of information. This content can take one of two forms:

1. a stream of bytes (accessed using a file metaphor)
2. a composite object (accessed using a graph metaphor)

Applications control whether the content of a particular node is of Type1 or of Type 2. Since Type 1 content obeys the standard file metaphor, it can be used to store the same types of information as files, e.g., text, bitmaps, line drawings, digitized audio and video, spreadsheets, and other binary data. Applications that can read and write conventional files can read and write Type 1 content with no changes. Type 1 content is stored with the node that contains it.

When a node has Type 2 content, then the content is stored *separately* as a composite object called a subgraph.[2] A *subgraph* is defined as a subset of the nodes and links in the artifact that is consistent with graph-theoretic constraints. For example, all subgraphs satisfy the condition that if a link belongs to a subgraph, then so do the link's source node and target node. Nodes and links may belong to multiple subgraphs at the same time, but every node and link must belong to at least one subgraph. Our data model also provides *strongly typed* subgraphs (e.g., trees and lists) that are guaranteed to be consistent with their type.

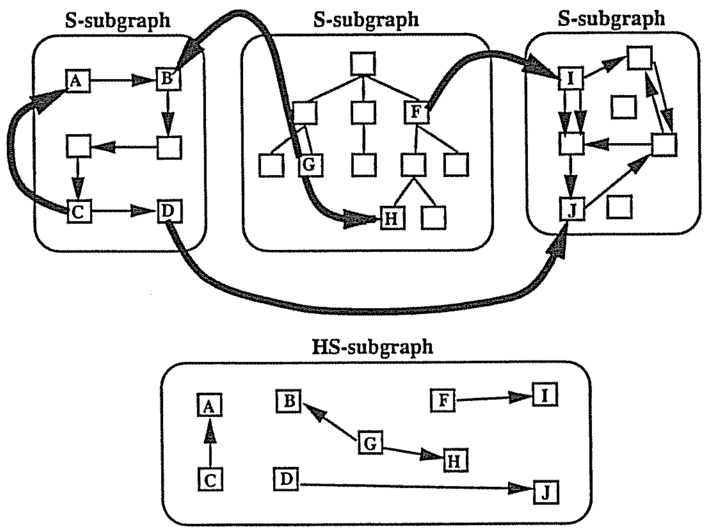

Figure 1: Examples of Hyper-structural Linking

Links can have both attributes and content associated with them. Moreover, the data model defines two classes of links: structural and hyper-structural. Structural links (S-links) are used to store the essential structure of an artifact. By contrast, hyper-structural links (HS-links) are lighter-weight objects that represent relationships that cut across

[1]Links to links are prohibited.

[2]Hereafter, Type 1 content will be referred to as *file content* and Type 2 content will be called *subgraph content*.

the basic structure (see Figure 1). Subgraphs containing only structural links are called S-subgraphs; those containing hyper-structural links are called HS-subgraphs.

USING THE DATA MODEL TO ORGANIZE INFORMATION

The data model encourages users to compose a large artifact from small subgraphs using subgraph content. This organization can improve human comprehension of the artifact and increase the potential for concurrent access to individual components. The best way to understand these mechanisms is by example.

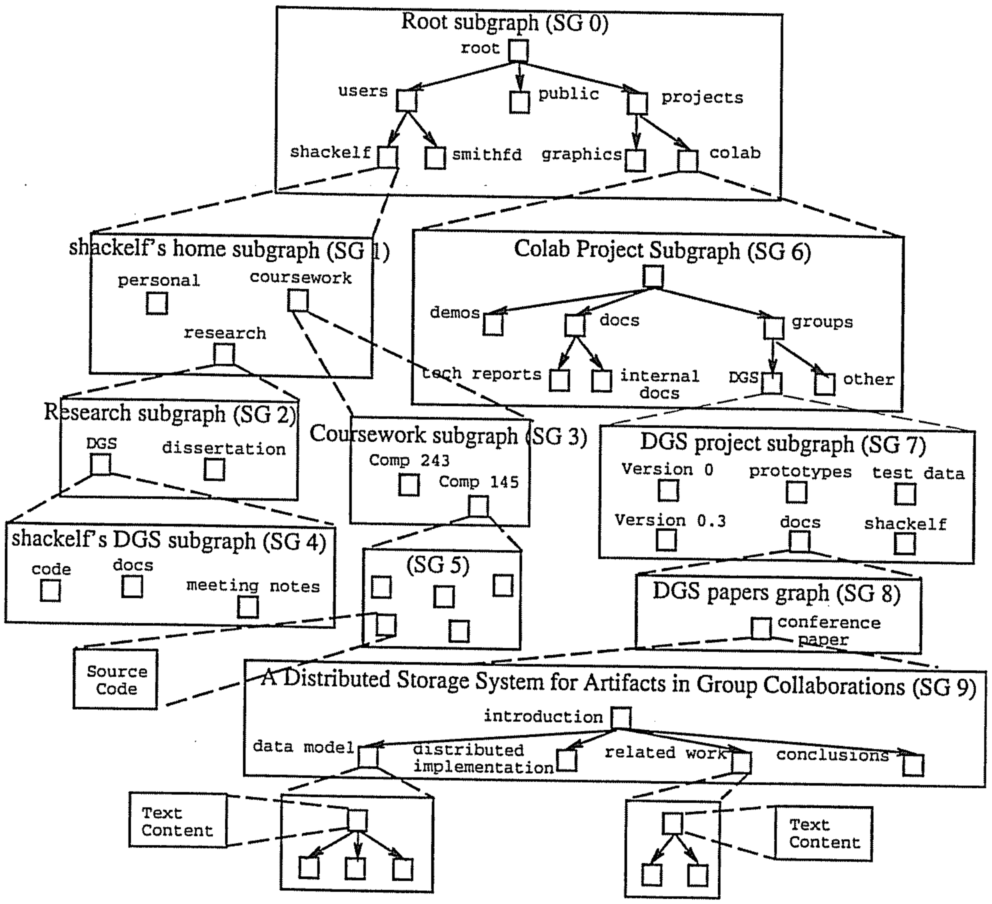

Figure 2: Organizing the Public and Private Pieces of an Artifact

Figure 2 illustrates one way to organize the public and private materials associated with a large research project (node content is indicated by dashed lines). One can observe that Figure 2 subsumes the organization of data in a conventional file system while providing additional mechanisms for storing meta-information about files (in attributes) and for representing semantic and structural relationships between files (in links).

Subgraph *SG 9* in Figure 2 is the top-level subgraph of a document. A useful exercise is to compare this graph structure with the way that the conference paper would be stored in a conventional file system. The most striking difference is the number and size of the nodes that compose the document. Whereas a conventional document would normally be stored in a single file or a small number of files, the DGS data model encourages a user to divide documents into many smaller nodes and subgraphs. This maximizes the benefits of hyper-structural linking because each node expresses a single concept or idea. By dividing a document into different subgraphs, collaborators may be able to structure their materials for easier concurrent access.

FINE-GRAINED LINKING USING ANCHORS

Although nodes are finer-grained than traditional files, there are still times when one would like to reference information at an even finer level. For example, an application might want to create a link that points to a specific word within a node, rather than to the node itself. To achieve fine-grained linking like this, 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, or an element of a line drawing. An anchor can be used to focus an HS-link onto a specific place within the content of a node. When an HS-link is paired with one or more anchors in its source or target nodes, it is called an *anchored HS-link.* The relationship between anchors and HS-links is many-to-many.

COMMON ATTRIBUTES AND GRAPH ATTRIBUTES

Some attributes are called *common attributes* because their values are independent of the context from which they are accessed. All objects---nodes, links, and subgraphs---can have common attributes. In addition, nodes and links can have context-sensitive attributes whose value may be different depending on the context from which they are accessed. This second type of attribute is called a *graph attribute* because a subgraph provides the context.

DESIGN AND IMPLEMENTATION

SYSTEM ARCHITECTURE

As shown in Figure 3, the DGS has a layered architecture that can be configured in a number of different ways. The *Application Layer* contains the user interface and other code that is application-specific. The top layer of the DGS is the *Application Programming Interface (API)* which exports a graph-oriented data model to applications. An overview of this data model was presented in a previous section. Most of the DGS is implemented in the bottom two layers: the Graph-Cache Manager (GCM) and the Storage Layer. The *GCM* implements the data model and performs local caching; the *Storage Layer* is responsible for permanently storing results.

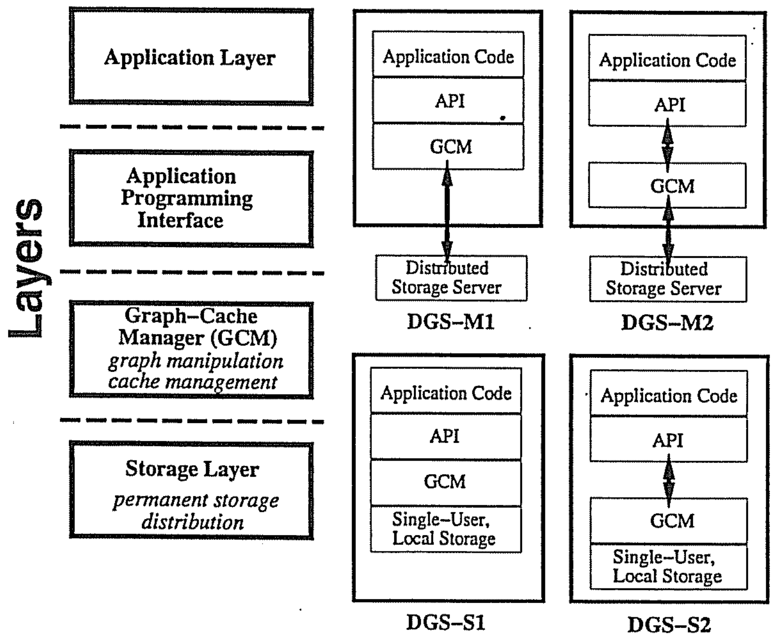

Figure 3: Four Implementations of the DGS Layered Architecture

Since the API isolates the application from the rest of the DGS, application code is portable across different implementations of the bottom two layers. We currently support two different implementations of the storage layer and two different methods for connecting the API with the GCM. This yields the four implementations that are shown in Figure 3. In DGS-M2, the application and the GCM run in different processes on the same machine; the Storage Layer is implemented as a multi-user, distributed storage server. DGS-M1 is the same except that the GCM is linked with the application to become a single process. The advantage of this design is better local response time due to reduced Inter-Process Communication (IPC). A disadvantage is that it increases the size of application executables. DGS-S1 and DGS-S2 follow a similar pattern except that the distributed storage server is replaced by a single-user, non-distributed storage layer.

THE OBJECT-ORIENTED API

The API for the DGS is a C++ class library (for a complete description, see (Shackelford, 1993)). Figure 4 shows the major classes in the inheritance hierarchy. The class Object defines operations that are common to all objects such as the functions for manipulating the common attributes of an object. Subclasses inherit the API of their parent class and extend the inherited API with more specialized functions.

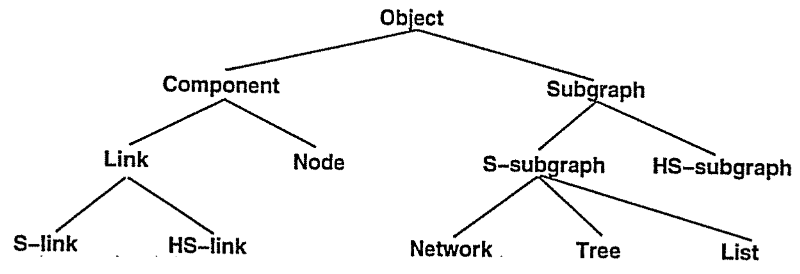

Figure 4: API Class Hierarchy

All node, link, and subgraph objects are identified by an object identifier (OID) that is universal and unique. Once an object is created by the DGS, its OID is never changed and the value is never reused even if the object is deleted. To applications, an OID is an "opaque" (uninterpreted) key that can be used to retrieve the corresponding object. However, we discourage application programmers from making direct reference to OIDs. Most operations can be performed without even knowing that OIDs exist.

CONCURRENT ACCESS TO OBJECTS

Since the DGS data model is object-oriented, the objects of the data model---nodes, links, and subgraphs---exist as distinct entities within the storage system. Before a user's application can access the data within a particular object (see Figure 5), the application must explicitly open the object using its Open() function.

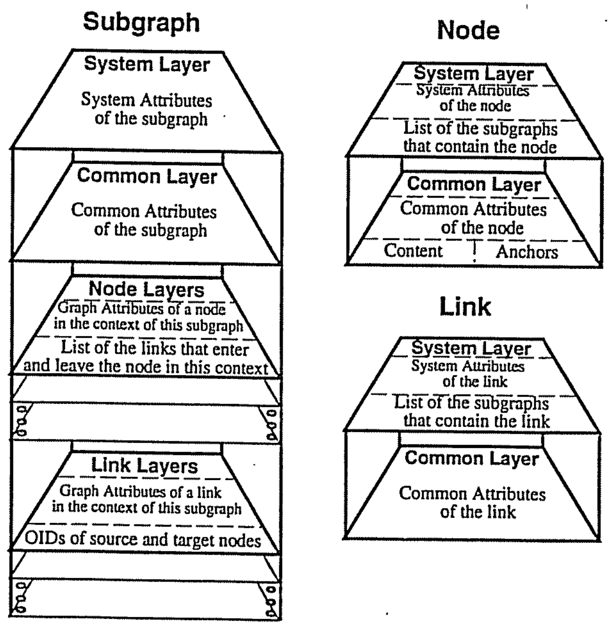
Subgraph
System Layer
System Attributes of the subgraph
Common Layer
Common Attributes of the subgraph
Node Layers
Graph Attributes of a node in the context of this subgraph
List of the links that enter and leave the node in this context
Link Layers
Graph Attributes of a link in the context of this subgraph
OIDs of source and target nodes
Node
System Layer
System Attributes of the node
List of the subgraphs that contain the node
Common Layer
Common Attributes of the node
Content
Anchors
Link
System Layer
System Attributes of the link
List of the subgraphs that contain the link
Common Layer
Common Attributes of the link

Figure 5: Information Stored in Nodes, Links, and Subgraphs

Open() will fail if the user lacks the proper access authorizations and if the request is in conflict with other requests in progress. Conflict can occur when different users try to access the same object concurrently. To specify allowable concurrent accesses, the API defines three access modes for nodes, links, and subgraphs: DGS_READ, DGS_WRITE, and DGS_READ_NO_ANCHOR. Applications must specify one of these modes as a parameter to Open(). DGS_READ allows operations that do not change subgraph membership, linking information, or attribute or content values. In the case of nodes, DGS_READ also allows anchor creation and deletion, but only when the application has read_write authorization on the HS-link that is being anchored. DGS_READ_NO_ANCHOR is defined only for nodes and allows all operations of DGS_READ except anchor creation and deletion. DGS_WRITE allows all operations.

The following rules govern concurrent access to an object:

- For links and subgraphs, multiple opens with DGS_READ access and a single open with DGS_WRITE access are allowed concurrently (as is the weaker case of multiple DGS_READ opens alone).
- For nodes, multiple opens with DGS_READ_NO_ANCHOR access and a single open with DGS_WRITE access are allowed concurrently (as is the weaker case of multiple DGS_READ and/or DGS_READ_NO_ANCHOR opens alone).

Thus, for nodes the design supports multiple non-annotating readers and a single writer OR multiple annotating readers. A consequence of this is that a writer is blocked from accessing a node that is being annotated by a reader and vice versa. 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.

ACCESS CONTROL FOR OBJECTS

Groups can control access to parts of the artifact by specifying access authorizations for node, link, and subgraph objects. Authorizations are expressed in an access control list that is stored with each object. An *access control list* maps names of users or groups of users to categories of operations that they are allowed to perform on the associated object. Two categories of authorizations are defined: access and administer. *Access authorizations* give users permission to access the data associated with a particular object. *Administer authorizations* give users permission to perform operations such as changing the object's access control list. Although the API does not define an explicit annotate permission, a similar effect can be accomplished by restricting the access authorizations associated with HS-subgraphs.

DISTRIBUTED IMPLEMENTATION

In this section we discuss the distributed implementations (DGS-M1 and DGS-M2 in Figure 3) with emphasis on key design decisions.

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 et al., 1991), (Kistler and Satyanarayanan, 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 AFS (Howard et al., 1988), Sprite (Nelson et al., 1988), and Coda (Kistler and Satyanarayanan, 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.

The basic structure of the system is shown in Figure 6. 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 subgraph 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-subgraphs. 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.

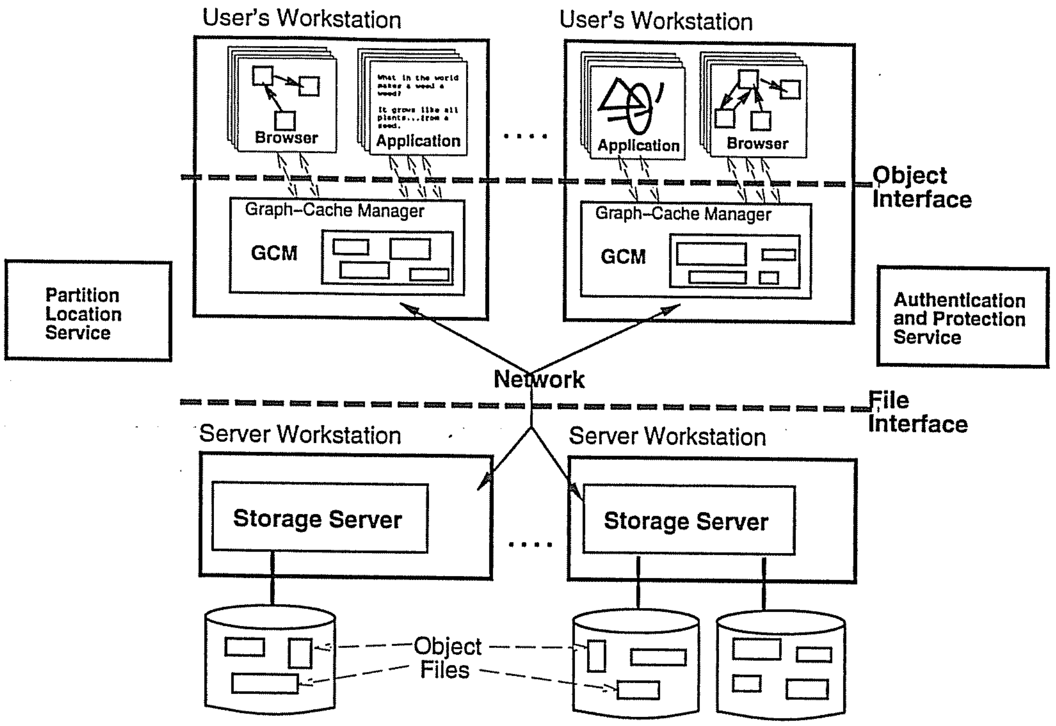

Figure 6: DGS System Structure

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 subgraph, or a group of links. An important performance optimization is that context-dependent attributes (graph attributes) and link information for all nodes in a subgraph are stored in one subgraph file. Thus, all of the data needed by a browser to display a subgraph is available from a single request (open) to the storage server. The structure of each type of file is shown in Figure 7. Nodes and subgraphs are stored individually, whereas links are grouped according to the subgraph in which they were created.

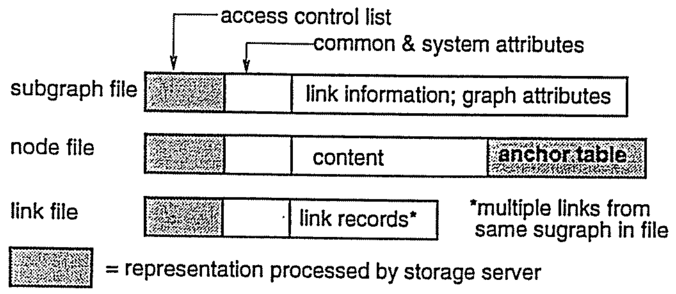
access control list
common & system attributes
subgraph file
link information; graph attributes
node file
content
anchor table
link file
link records*
*multiple links from same sugraph in file
= representation processed by storage server

Figure 7: 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, 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 subgraphs 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.

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.

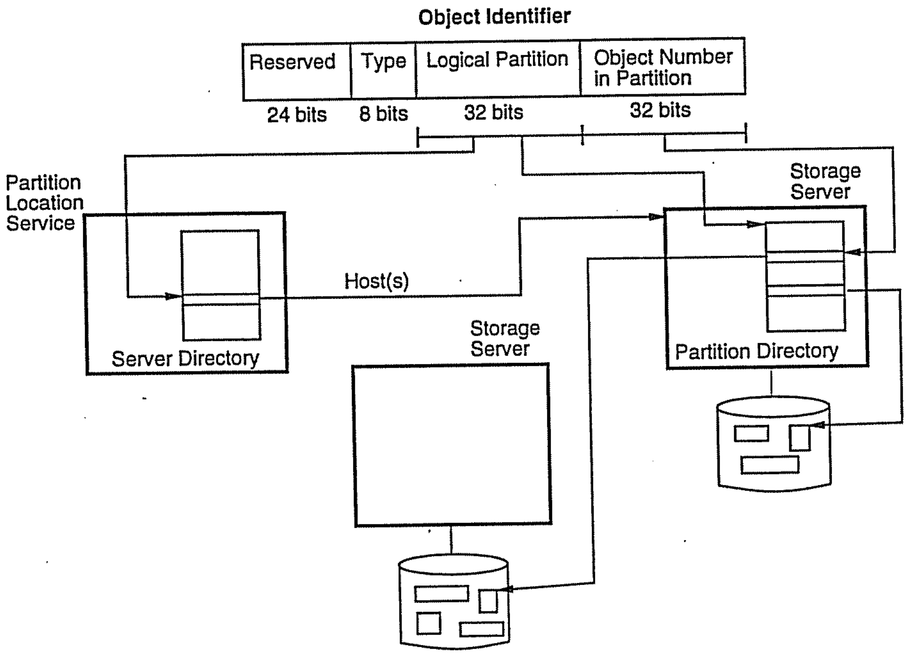

Figure 8: OID and Object Location

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 and Birman, 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. 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 AFS (Howard et al., 1988) and Sprite (Nelson et al., 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 synthetic browser on several workstations. This program has parameters that can be used to produce a wide range of browsing behaviors. In our first experiments we are using parameter values that represent the observed behavior of human subjects in a series of experiments we 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 subgraphs, 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 synthetic browser program. All workstations were connected by a single ethernet segment. 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.

The results show that one server can support at least 50 users. More extensive benchmark experiments are underway to validate this conclusion for a variety of configurations.

We are currently using the DGS 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 Fall 1993.

COMPARISON WITH RELATED WORK

In this section, we compare our design with several hypertext systems that have significant capability for supporting collaborating groups, i.e., Intermedia(Haan et al., 1992; Yankelovich et al., 1988), HyperBase/CHS(Schütt and Streitz, 1990; Schütt and Haake, 1993), Augment(Engelbart, 1984), Telesophy(Caplinger, 1987; Schatz, 1987), KMS(Akscyn et al., 1988), and HAM(Campbell and Goodman, 1988; Delisle and Schwartz, 1987). These systems differ widely on factors such as the data model supported, scalability, concurrent reader/writer semantics, and protection.

DGS, HyperBase/CHS, and Dexter(Halasz and Schwartz, 1990) support rich data models that include aggregates (named groups of objects), aggregates of aggregates, and aggregates as endpoints of links. Intermedia, HAM, and Augment do not use aggregates in composition or linking. Telesophy's data model has aggregates but does not give first-class status to links. HB1(Schnase et al., 1991) and Trellis(Stotts and Furuta, 1989) provide strong support for computation within hypertext but do not have aggregates. The DGS data model benefits from the graph-theoretical metaphor on which it is based and is the only system to provide 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 the DGS have made scalability a central issue in their designs. However, the DGS provides more flexibility in its data model and stronger consistency semantics.

Hypermedia System	Concurrent Reader/Writer Semantics	Protection of Objects
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
HyperBase/CHS	Activity markers are provided to warn applications of concurrent activity, but these markers are advisory in nature. All applications are notified when data is changed, so that they can update their view (if desired).	Access control will be based on user roles such as "manager" and "secretary" (not yet implemented).
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.
KMS	Uses an optimistic concurrency method. When a writer attempts to save a node, he/she 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.
Telesophy	Supports multiple concurrent readers and writers. When writes overlap, the last writer completely overwrites the work of others	could not be determined
DGS	Supports multiple non-annotating readers and a single writer OR multiple 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 administer permissions. Rather than associate a single annotate permission with a node, the DGS provides a more flexible mechanism of associating annotate permission with the HS-subgraphs that 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

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 and 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.

We observe that most of the academic research in hypermedia is not based on the sort of modular architecture that was described at the beginning of this paper. Although many communities view hypermedia as an "interesting" application, we take the perspective (also expressed in (Schnase et al., 1991)) that hypermedia has a broader role to play. In our opinion, hypermedia is not just an application, but is a new paradigm for the way we work and collaborate with each other. As such, it will be an essential component of the next generation of operating system support. Our experiences with DGS strongly indicate that it is possible to achieve the richer functions needed for hypermedia storage with cost, performance, and scalability comparable to the best conventional distributed file systems (e.g., AFS).

As we look to the future, additional issues we will explore pertain to wide-area network access, dynamic change notification, graph traversal, and support of a richer set of graph and set operations and queries. Many of these extensions lend themselves to the sort of modular approach that is suggested in the Strawman Reference Model (Thompson, 1990).

REFERENCES

Akscyn, R. M., D. L. McCracken, and E. A.Yoder (1988, July). KMS: A distributed hypermedia system for managing knowledge in organizations. Communications of the ACM 31(7), 820-835.

Baker, M. G., J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout (1991, October). Measurements of a distributed file system. Operating Systems Review, Special Issue: Proceedings of the 13th ACM Symposium on Operating Systems Principles (Pacific Grove, CA) 25(5), 198-212.

Campbell, B. and J. M. Goodman (1988). HAM: A general purpose hypertext abstract machine. Communications of the ACM 31(7), 856-861.

Caplinger, M. (1987, October). An information system based on distributed objects. In OOPSLA '87 Proceedings, pp. 126-137.

Delisle, N. M. and M. D. Schwartz(1987, April). Contexts: a partitioning concept for hypertext. ACM Transactions on Office Information Systems 5(2), 168-186.

Engelbart, D. C.(1984, February). Authorship provisions in AUGMENT. In Proceedings of the 1984 COMPCON Conference, San Franscisco, CA, pp. 465-472.

Haan, B. J., P. Kahn, V. A. Riley, J. H. Coombs, and N. K. Meyrowitz (1992, January). IRIS hypermedia services. Communications of the ACM 35(1), 36-51.

Halasz, F. and M. Schwartz(1990). The Dexter hypertext reference model. In Proceedings of the NIST Hypertext Standardization Workshop (Gaithersburg, Maryland), pp.1-39.

Howard, J. H., M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West (1988, February). Scale and performance in a distributed file system. ACM Transactions on Computer Systems 6(1), 51-81.

Jeffay, K., J. K. Lin, J. Menges, F. D. Smith, and J. B. Smith (1992). Architecture of the artifact-based collaboration system matrix. In Proceedings of ACM CSCW '92 Conference on Computer-Supported Cooperative Work, CSCW Architectures, pp. 195-202.

Joseph, T. A. and K. P. Birman(1986, February). Low cost management of replicated data in fault-tolerant distributed systems. ACM Transactions on Computer Systems 4(1), 54-70.

Kistler, J. J. and M. Satyanarayanan(1991, October). Disconnected operation in the Coda file system. Operating Systems Review, Special Issue: Proceedings of the 13th ACM Symposium on Operating Systems Principles (Pacific Grove, CA), 25(5), 213-225.

Nelson, M. N., B. B. Welch, and J. K. Ousterhout (1988, February). Caching in the Sprite network file system. ACM Transactions on Computer Systems, 6(1), 134-154.

Schatz, B. R.(1987). Telesophy: A system for manipulating the knowledge of a community. In Proceedings of Globecom '87, New York, pp. 1181-1186. ACM.

Schnase, J. L., J. J. Leggett, and D. L. Hicks (1991, October). HB1: Initial design and implementation of a hyperbase management system. Technical Report TAMU-HRL 91-003, Hypertext Research Lab, Texas A&M University.

Schütt, H. and J. M. Haake (1993, March). Server support for cooperative hypermedia systems. In Hypermedia '93, Zurich.

Schütt, H. A. and N. A. Streitz (1990). Hyperbase: A hypermedia engine based on a relational database management system. In Proceedings of the ECHT'90 European Conference on Hypertext, Databases, Indices and Normative Knowledge, pp.95-108.

Shackelford, D. E. (1993, January). The Distributed Graph Storage System: A users manual for application programmers. Technical Report TR93-003, Department of Computer Science, The University of North Carolina at Chapel Hill.

Smith, D. K. (1992). Hypermedia vs.paper: User strategies in browsing SNA materials. Technical Report TR92-036, Department of Computer Science, The University of North Carolina at Chapel Hill.

Smith, J. B. and F. D. Smith (1991). ABC: A hypermedia system for artifact-based collaboration. In Proceedings of ACM Hypertext'91, Construction and Authoring, pp. 179-192.

Stotts, P. D. and R. Furuta (1989). Petri-net-based hypertext: Document structure with browsing semantics. ACM Transactions on Information Systems 7(1), 3-29.

Thompson, C. W. (1990, January). Strawman reference model for hypermedia systems. In Proceedings of the NIST Hypertext Standardization Workshop (Gaithersburg, Maryland), pp. 189-196.

Yankelovich, N. et al. (1988, January). Intermedia: The concept and the construction of a seamless information environment. IEEE Computer 21(1), 81-96.

ACKNOWLEDGMENTS

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