Requirements Document for the UNC Distributed Graph Service

TR91-051
December, 1991

Douglas E. Shackelford

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

Portion of this research were supported by the National Science Foundation (Grants # IRI-9015443 and the IBM Coporation (SUR Agreement #866), and The Office of Naval Research (Contract # N00014-86-K-0680).

UNC is an Equal Opportunity/Affirmative Action Institution.

Abstract

A distributed hypermedia storage service--called the UNC Distributed Graph Service (DGS)--will become the database for a hypermedia computer-supported, cooperative work environment called ABC. The DGS data model has been designed to support the unique storage needs of users who are collaborating to solve complex problems. Since it is based on the formalism of directed graphs, the DGS data model provides a structured method for organizing and relating the diverse materials that collaboration produces. In our discussion, we will first enumerate the special requirements that a collaborative environment places on a database and then describe specific mechanisms for satisfying those requirements in the DGS data model. A number of research issues are raised, including: partitioning and structuring the materials produced by collaboration; distinguishing between public and private views of information; providing an extensible, open architecture for applications; representing fine-grained associations using anchored hyperlinks; providing appropriate concurrent access to information; and maintaining the integrity, consistency, and completeness of the database as a whole.

Acknowledgements

This work was done mostly during the Spring and Summer of 1991 and is the product of a series of discussions led by the author, John B. Smith, and F. Donelson Smith. Other participants included Joan Boone, John M. Hilgedick, Jin-Kun Lin, John Menges, Michael P. Wagner, and Zhenxin Wang. In addition, the author would like to acknowledge contributions by Murray Anderegg and Barry Elledge on an earlier design. Major support has come from NSF (Grant #IRI-9015443) and the IBM Corporation (SUR Agreement #866), with additional support from ONR (Contract #N00014-86-K-00680).

Table of Contents

Section 1. Introduction 1

Section 2. An Overview of the DGS Data Model and Its Use 3

2.1. Introduction 3

2.2. Using Graph-As-Content Relationships to Partition Data 4

2.3. Using Hyperlinks and Anchors 7

2.4. Summary 9

Section 3. The Terminology of the DGS Data Model 10

3.1. Introduction 10

3.2. Distinguishing Structure from Hyper-Structure 10

3.3. A Glossary of the Terms Used in the DGS Data Model 10

3.3.1. Node 10
3.3.2. Link 11
3.3.2.1. Structural Link 11
3.3.2.2. Hyperlink 11
3.3.3. Graph 11
3.3.3.1. Subgraph 11
3.3.3.2. Hypergraph 12
3.3.4. Attribute 12
3.3.4.1. Graph Attribute 12
3.3.4.2. Common Attribute 12
3.3.5. Temporary and Permanent Graphs 12
3.3.5.1 Temporary Graph 12
3.3.5.2. Permanent Graph 13
3.3.6. Content, Anchors, and Hyperlinks 13
3.3.6.1 Content of Nodes and Links 13
3.3.6.2. Anchors 13
3.3.6.3. Anchored Hyperlinks 13

Section 4. Synchronization of Collaborating Users 15

4.1. Introduction 15

4.2. Synchronous Collaboration 15

4.3. Asynchronous Collaboration between Users whose Updates Overlap in Time but not in Space 15

4.4. Asynchronous Collaboration between Users whose Updates Overlap in both Space and Time 16

4.4.1. Introduction 16
4.4.2. Distance Between Overlapping Asynchronous Applications 16
4.4.3. Access Mode of Applications 16
4.4.4 Granularity of Visible Updates 17
4.4.5. Frequency with which Independent Views of the Same Object are Synchronized 17

4.5. Synchronization in the UNC Distributed Graph Service 17

Section 5. Operations on Graph Objects 18
5.1. Introduction 18
5.2. Concurrent Access 18
5.2.1. Categories of Accessible Information 18
5.2.2. Concurrent Access to System Data 18
5.2.3. Concurrent Access to Object Data and Graph Data 18
5.2.3.1. Operations for Accessing Object and Graph Data 18
5.2.3.2. Recovery from Application Failures 19
5.2.3.3. Undo Semantics 19
5.2.4 Concurrent Access to Anchor Tables 20
5.3. Permissions and Access Control 20
5.4. Anchor-Related Operations 21
5.5. The Creation of Public and Private Views 22
5.6. Operation Completion Semantics 25
Section 6. References 26

Section 1. Introduction

The Artifact-Based Collaboration (ABC) environment is being developed to support groups of people as they collaborate to solve complex intellectual problems. ABC will include a set of specialized applications, a shared window conferencing facility, real-time audio/video, and a hypermedia storage service. We are developing ABC's hypermedia storage service which we call the UNC Distributed Graph Service (DGS). In this report, we describe the requirements that we have established for the DGS and a data model that satisfies those requirements. The data model is the basis for the implementation of the DGS which is in progress.

The DGS will store the materials which are created when a group of people collaborate to solve a complex intellectual problem such as the development of a large software system. The size and complexity of these tasks require that they be divided into sub-problems which can be solved by different people working simultaneously. To facilitate communication among the people involved in different aspects of the problem, groups generate a variety of interrelated artifacts. For example, software development generates artifacts such as concept papers, architecture, requirements, specifications, programs, diagrams, reference and user manuals, and administrative documents. These artifacts are created by the group for its own use and as a part of the final product.

Hypermedia is a natural tool for storing and organizing the artifacts of group collaboration, since it can represent fine-grained associations between artifacts. Hypermedia applications divide artifacts into nodes which are related to each other by explicit node-to-node links. For example, a document is an artifact which can be partitioned into nodes which represent chapters, sections, and paragraphs. Hypermedia links can be used to define the structural relationships between nodes within a document as well as semantic relationships between nodes in different documents.

In addition to storing public artifacts such as requirements and specifications, hypermedia is used to store private artifacts which are created by individuals for their own use. Examples of private artifacts include personal notes and correspondence. Often, private artifacts contain important information about the process of software development which is absent from the final product. For instance, notes from an early design meeting can hold the key to understanding an important design decision. Private artifacts are also invaluable to researchers who study the process of group collaboration. Unfortunately, private artifacts are often discarded or lost. In a hypermedia environment, these private artifacts can be preserved by linking them to more permanent artifacts.

Another advantage of hypermedia environments is that they can support the work patterns of a group more naturally. Since collaborators must work together, a collaborative database should provide mechanisms which allow them to share artifacts. This requirement is complicated by the need for simultaneous access and also by the fact that people need to work with the same artifact for hours or even days at a time. In this regard, traditional transaction-oriented databases fall short, since they are designed to support brief, disjoint jobs. However, users of hypermedia databases can reduce this problem by partitioning their databases according to the most common access patterns.

By carefully partitioning a database, one can also improve its ability to support progressively larger amounts of data and numbers of users. When a database can be partitioned, it is easier to increase the capacity of the system by distributing the database across a group of computers and file systems. In addition, performance is improved when

data is stored near the people who need it. This is especially important when users are widely distributed geographically.

As the amount of data increases, partitioning can also make it easier for users to find and comprehend information in the database. When hypermedia structures become too large or too complex, human beings 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 large hypermedia structures. Once the database has been partitioned into smaller structures, the pieces can be understood more easily and then related to each other via cross-structural links.

A final requirement for a hypermedia database for collaboration is that it provide a relatively open architecture. In particular, existing applications should fit easily into the collaboration environment. The environment should support both special purpose hypermedia applications as well as general purpose file-oriented programs such as text editors and spreadsheets. Although much of this responsibility is beyond the mandate of the hypermedia storage service, it must provide primitive support for different types of data and applications.

In the discussion that follows, we describe the data model that the DGS will provide. In Section 2, we give an informal overview of the DGS data model and methods for using the model to structure information. Section 3 precisely defines terminology and describes the data model more rigorously. Sections 4 and 5 provide details of how the data model functions.

Section 2. An Overview of the DGS Data Model and Its Use

2.1. Introduction

In this section we describe the fundamental concepts of the UNC Distributed Graph Service (DGS) data model. Our intent is to provide an overview of the basic constructs and their uses. A more complete definition of the data model appears in the next section.

The basic objects in the data model are nodes, links, and typed graphs, all of which can have attributes attached to them. An attribute is a user-defined variable which can be of arbitrary type and size. A graph consists of a set of nodes and links. Graphs may contain cycles and can even be disconnected (see Figure 1).

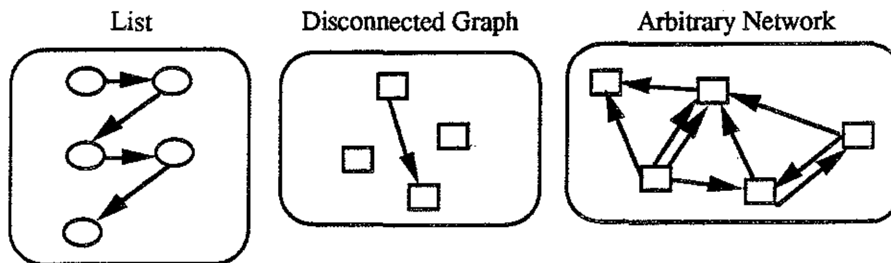


Figure 1: Three examples of graphs

In addition to normal user-defined attributes, each node and link has a predefined variable called *content*, which is a pointer to a file or a graph. The DGS will provide the same protection to the content of a node or link as it does to the node or link itself. Thus, if a node is read-only, then its content will also be read-only. The association between a node or link and a file is called a file-as-content relationship. Similarly, content relationships involving graphs are called graph-as-content relationships. As demonstrated in Figure 2, content relationships are shown as dotted lines from a node to its content.

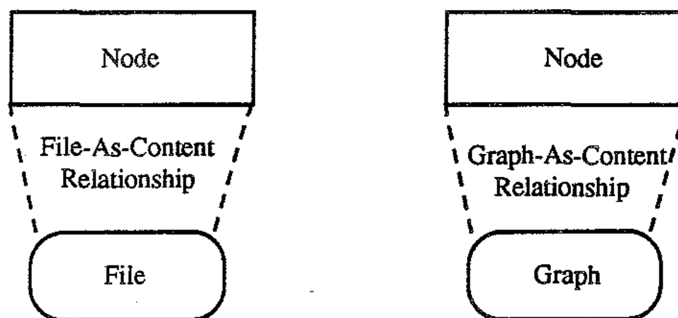


Figure 2: Dotted lines are used to indicate the two types of content relationships.

The file-as-content relationship is a natural method for integrating file-oriented applications into the hypermedia environment. For example, a file-oriented application such as a text editor can store its data in a file which is the content of a node. Once the file is stored as the content of a node, it can be linked to other information in the hypermedia database.

2.2. Using Graph-As-Content Relationships to Partition Data

The DGS data model also encourages users to decompose large data structures into small graphs which are related by graph-as-content relationships. For example, think about how a piece of software could be structured in this data model (see Figure 3). The whole program could be represented as a top level graph. Nodes within the program-graph could represent program modules. To take the example one level deeper, suppose that each module-node contained a different graph which we will call its module-graph. A module-graph could contain a node for each procedure within the module. At the lowest level, each procedure-node could contain a file of source code as its content. By structuring the program in this way, a user could create a natural partitioning which would improve human comprehension in addition to increasing the amount of concurrent access that would be possible.

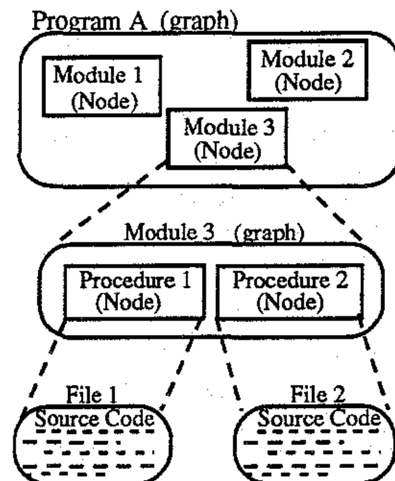


Figure 3: Using Content Relationships to Structure a Computer Program.

In fact, graph-as-content partitioning can be applied to a wide variety of problems. Many common structures, such as the organization of a document, can be represented naturally by trees. Unfortunately, large documents result in correspondingly large trees, which may be difficult to manipulate efficiently. Also, monolithic trees discourage users from working concurrently, since it may be difficult for them to work simultaneously on the same tree. Partitioning can reduce both of these problems. Under the DGS data model, a user can specify the first few levels of a document to the point where the leaves of a high level tree represent the chapters (see Figure 4). Then, the user can define a different graph as the content of each leaf node in the high level tree. The chapters themselves can be stored in these leaf graphs. Depending on the overall size of the document, this process can be repeated iteratively as needed. The resulting composite structure will be easier to use and understand than the original monolithic tree.

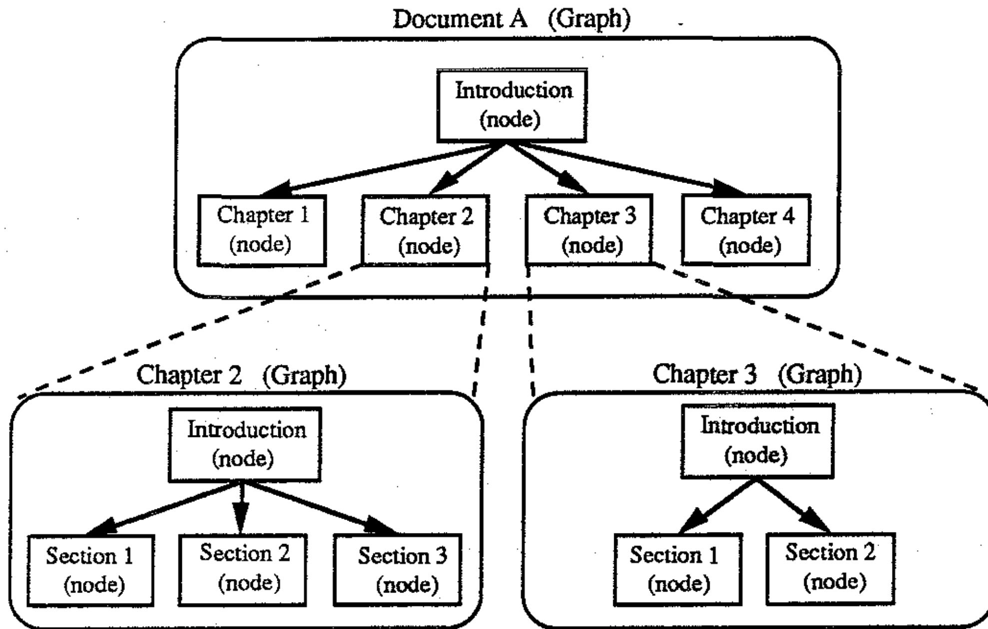


Figure 4: Using Content Relationships to Structure a Document.

Although graph-as-content partitioning works well with hierarchical structures, it is not limited to them. Structures involving graph-as-content relationships can have complex and even cyclic composite structures. Composite structures are used to integrate data which has been partitioned for practical reasons. For instance, the document in Figure 4 is a composite structure. The chapters are stored in different graphs, but are integrated into a single document by the graph-as-content relationships.

A composite structure is also an ideal vehicle for representing the global structure of a database. For example, we can designate a particular graph to be the *root graph* of the database. Given this, the database can be defined as the union of all of the graphs which are reachable from the root graph via any number of graph-as-content relationships. A surprising consequence of this is that we can combine two or more different databases simply by creating graph-as-content relationships to their root graphs. Thus, not only can a composite structure represent a single database, but it also can logically relate the pieces of a multi-database (see Figure 5).

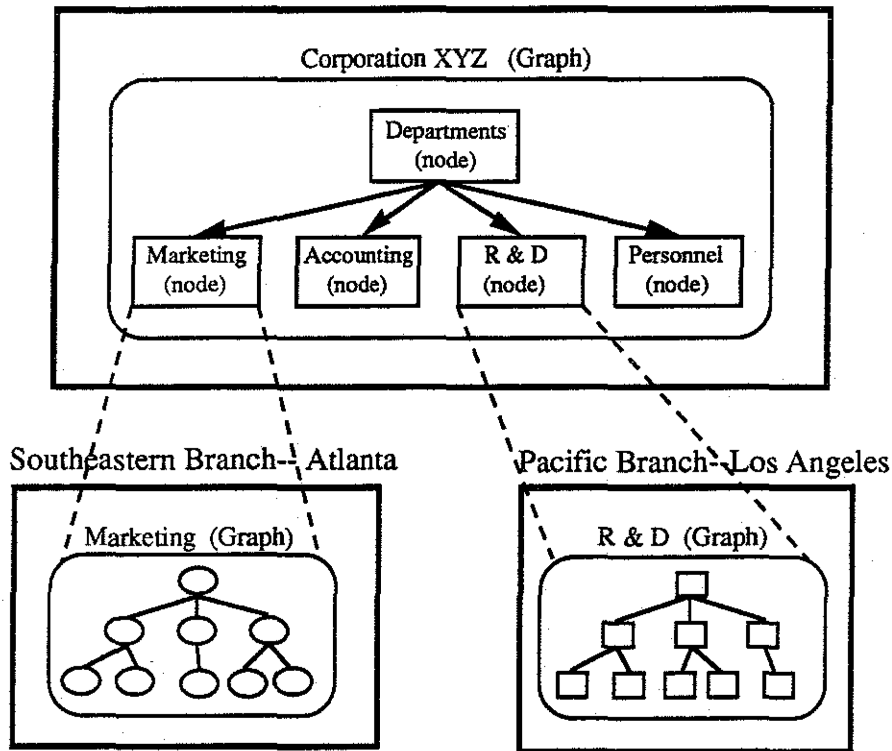


Figure 5: Using Content Relationships to Construct a Multi-Database which Combines Three Databases at Three Different Sites.

The natural structure for a root graph is the structural organization of the group which uses the database. For instance, the root graph of a corporate database might be a tree with leaves that represent the departments within the corporation, such as marketing, personnel, and research and development. Each of these leaves might contain the root graph of a departmental database. The decomposition could logically continue to the level of projects within the departments. Each project could be divided into group artifacts and artifacts belonging to individual employees. At the leaves of this immense composite graph would be the artifacts themselves: documents, diagrams, spreadsheets, programs, memos, etc. Despite the fact that this corporate multi-database might be partitioned at many levels, its composite structure would be remarkably coherent. Figure 6 is an example of how a composite structure could be used to store both public and private information at UNC. The root graph SG 0 distinguishes among private information belonging to users, private information belonging to groups, and public information. Once again, we see that the graph-as-content relationship is a powerful tool for organizing and combining information which, for practical reasons, must be partitioned.

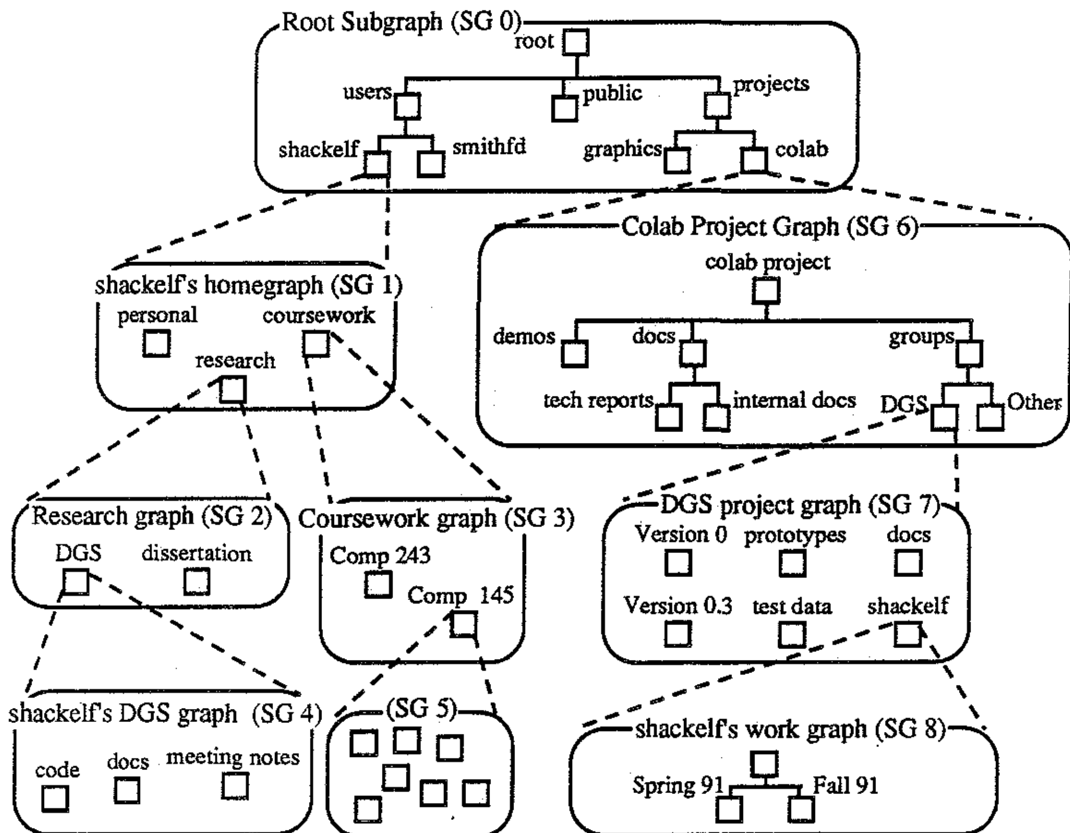


Figure 6: Using Content Relationships to Partition and Organize Public and Private Materials.

2.3. Using Hyperlinks and Anchors

However, despite the usefulness of composite structures, they are limited in their ability to capture associations which cut across the global structure. For instance, a graph-as-content relationship should not be used to represent the association between a piece of computer code and the document which describes its specification. Since this type of association crosses the normal structural boundaries, it cannot be represented naturally with structural relationships. Thus, the DGS data model provides a separate mechanism called the hyper-structural link or hyperlink. Hyperlinks are used to capture associations between nodes which are in different graphs. They also can capture hyper-structural relationships between nodes within the same graph, such as a reference in one chapter of a document to its appendix (see Figure 7).

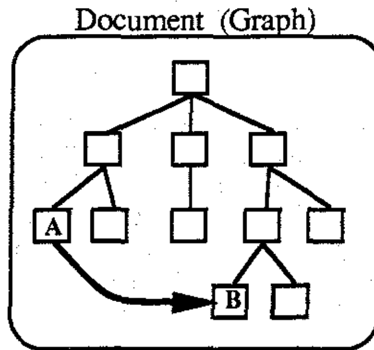


Figure 7: The Hyperlink from Node A to Node B Cuts Across the Structure of the Document

Although hyperlinks can represent relationships between whole nodes of information, they cannot distinguish finer-grained associations. For example, a group of users might use a node to store the glossary of terms which are common to their project. Given this, a user might want to create a link from the occurrence of a term in a document to its definition in the glossary node. Unfortunately, a hyperlink alone is insufficient. A hyperlink could link the two nodes which contain the term and the glossary, but it could not link the term itself to the definition. To achieve this finer-grained linking, the DGS data model provides the concept of an anchor within a node. An anchor refers to part of a node's content, such as a word in a paragraph or a definition inside of the glossary node. Links are associated with anchors through a table in the node. By associating an anchor with a hyperlink, a user can effectively extend the hyperlink into the content of a node, thereby narrowing the focus of the link (see Figure 8).

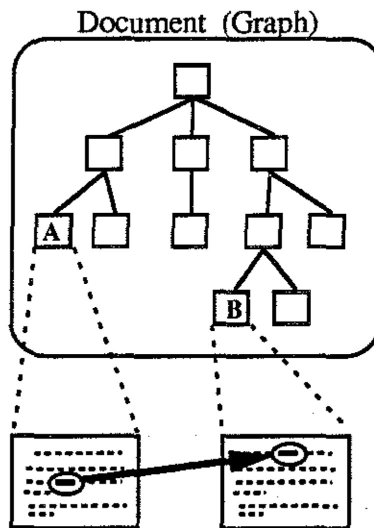


Figure 8: An Anchor Extends the Hyperlink into the Content of Each Node.

2.4. Summary

In this section, we introduced the fundamental constructs of the DGS data model. The basic objects in the data model are nodes, links, and graphs. In addition, nodes and links can engage in two kinds of content relationships: file-as-content and graph-as-content. The latter was shown to be a powerful method for partitioning hypermedia data into useful and understandable composite structures. Finally, we introduced hyperlinks and anchors as constructs for representing relationships which cut across the global structure. In the next section, we will precisely define the terminology of the DGS data model.

Section 3. The Terminology of the DGS Data Model

3.1. Introduction

The data model of the UNC Distributed Graph Service (DGS) is designed to support group collaboration in a hypermedia environment. In the previous section, we introduced the data model and discussed informally the concepts of node, link, graph, content, hyperlink, and anchor. In this section, we will extend these concepts by formally distinguishing basic structure from hyper-structure. This dichotomy will lead us to divide graphs into two types: structural graphs called subgraphs and hyper-structural graphs called hypergraphs. The section will conclude with the definitions of all the terms used in the construction of our data model.

3.2. Distinguishing Structure from Hyper-Structure

Much of hypermedia's power and flexibility comes from explicitly representing relationships that are normally implicit. In fact, structure is simply the explicit representation of essential implicit relationships. When a document is transformed into a tree, the implicit organization of the document becomes explicit in the structure of the tree. Similarly, the implicit organization of a department is explicit in the structure of a composite graph which relates the projects within the department. When structure is removed from an artifact or becomes inconsistent, then the meaning of the artifact must be either changed or destroyed. Thus, structure is a necessary and fundamental part of any artifact.

Nevertheless, some implicit relationships cannot be represented in structure. For example, collaborators commonly attach annotations to each other's work, but it is awkward to use structural relationships for this purpose. The annotation is not a part of the thing that is being annotated and, thus, should not participate in its structure. In the previous section, we introduced the hyperlink as a mechanism to capture these non-structural relationships. Like the structural link, the hyperlink is the explicit representation of an implicit concept, although its use and purpose is very different. Whereas structure is essential, hyper-structure is supplementary and thus subordinate. Because of this contrast, the DGS data model distinguishes explicitly between these two concepts. Links are described as being either structural links or hyperlinks. Similarly, graphs are categorized as either structural subgraphs or hypergraphs. By making this distinction, the data model explicitly recognizes the subtle differences between structural and hyper-structural information.

3.3. A Glossary of the Terms Used in the DGS Data Model

An important feature of the DGS data model is that it is object-oriented. The objects of the data model--nodes, links, and graphs--exist as distinct entities in the database. In the remainder of this section we will provide precise definitions for all of the terminology used in the DGS data model. In subsequent sections, we will provide details about how the data model functions.

3.3.1. Node

Nodes are repositories for information. Information is associated with a node via attributes defined on the node. In addition, each node has a special content variable which is a pointer to data which is external to the node. The majority of a node's information is stored in its *content* which can be either a file or a graph. Nodes may have arbitrary

numbers of in-coming and out-going links. A node can be contained in more than one graph at the same time, but every node must always be contained in at least one subgraph. A node ceases to exist when a user removes it from the last subgraph which contains it. When a user removes a node from a graph, then all incident links are removed automatically as well.

3.3.2. Link

A link represents a relationship between two nodes. These are referred to as the source node of the link and the target node of the link. Both the source node and the target node must exist. If a user removes either the source node or the target node of a link, then the link itself is removed automatically. Links have a direction, although traversal is supported in either direction. Like nodes, links are repositories for the information which is stored in their attributes and content. A link can be contained in more than one graph at the same time. The two types of links are structural links and hyperlinks. Throughout this document "link" is used to mean "both structural links and hyperlinks."

3.3.2.1. Structural Link

Structural links are used to represent the backbone of information such as the structure within a document. Structural links relate nodes within a subgraph. A structural link must always be contained in at least one subgraph but can never be contained in a hypergraph. A structural link ceases to exist when a user removes it from the last subgraph which contains it. (see also *link*)

3.3.2.2. Hyperlink

A hyperlink is a non-structural link which connects two nodes which may or may not be in the same subgraph. Hyperlinks are used to link across the structure within a given subgraph and also to link nodes in different subgraphs. For example, a hyperlink could represent a cross-reference within a document, or it could take the user from a word in one document to the definition of the word in another document. A hyperlink must always be contained in at least one hypergraph but can never be contained in a subgraph. A hyperlink ceases to exist when a user removes it from the last hypergraph which contains it. (see also *link*)

3.3.3. Graph

A graph is a set of nodes and links. Nodes and links can belong to more than one graph simultaneously, but every node and link must be a member of at least one graph. 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. Small graphs have the additional advantage that they can be stored and retrieved more efficiently. Information which is relevant to a whole graph can be stored in user-defined attributes in the graph object itself. The two types of graphs are subgraphs and hypergraphs. Throughout this document "graph" is used to mean "both subgraphs and hypergraphs".

3.3.3.1. Subgraph

Subgraphs organize and name groups of related information. A subgraph is a set of structural links and a set of nodes. The DGS data model constrains the set of nodes in a graph to be a superset of the set of nodes which are either the source or target of a link in

the graph. In addition to the attributes which are stored in nodes and structural links, attributes can also be stored in the subgraph object itself. Different subgraphs may share nodes and links. When a user removes a node or link from a particular subgraph, it does not affect its membership in other subgraphs. In addition to the basic subgraph, the DGS data model also provides a predefined set of typed subgraphs such as trees and lists. The DGS will guarantee that typed subgraphs are always in a state which is consistent with their type. For example, the DGS will not allow a user to create a cycle in a subgraph of type *tree.* (see also *graph*)

3.3.3.2. Hypergraph

Hypergraphs name groups of non-structural links called hyperlinks. A hypergraph is a set of hyperlinks and nodes such that the set of nodes is exactly equal to the set of all the hyperlinks' source and target nodes. In addition to nodes and hyperlinks, hypergraphs also store information in user-defined attributes in the hypergraph object itself. Different hypergraphs may share nodes and hyperlinks. When a user removes a node or hyperlink from a particular hypergraph, it does not affect its membership in other hypergraphs. (see also *graph*)

3.3.4. Attribute

Attributes are user-defined, named variables which are attached to graphs, nodes, and links. Each graph, node, and link supports an arbitrary number of user-defined attributes. Attribute values can be of arbitrary size and type.

3.3.4.1. Graph Attribute

A graph attribute is an attribute which is defined in a node or a link, but which is strongly associated with a particular graph which contains the node or link. For example, a node called Node A could use a graph attribute to stores its (X,Y) screen position within a subgraph called Subgraph A. Because the (X,Y) position would be relative to the positions of other nodes in Subgraph A, the attribute would not be meaningful when users look at other subgraphs that contain Node A. The attribute would have meaning only when a user has opened Subgraph A. Thus, to access a graph attribute in a node or link, a user must have first gained access to the graph which is associated with the attribute. (see also *attribute*)

3.3.4.2. Common Attribute

Unlike graph attributes, common attributes can be defined in graphs as well as nodes and links. Also, common attributes have a broader scope than graph attributes, since common attributes are meaningful in all contexts. Nodes and links retain their common attributes as they are copied from graph to graph. Therefore, if a node is expected to have the same title in each graph that contains it, then the title should be stored as a common attribute of the node. To access a common attribute, a user must have first gained access to the node, link, or graph which contains the attribute. (see also *attribute*)

3.3.5. Temporary and Permanent Graphs

3.3.5.1 Temporary Graph

A temporary graph is a graph which holds an intermediate result, such as the intersection of two subgraphs. A temporary graph may be created implicitly by the DGS or explicitly by

the user. Temporary graphs are automatically destroyed at the end of the session in which they were created.

3.3.5.2. Permanent Graph

A permanent graph is a graph which is created explicitly by a user. A permanent graph exists until it is explicitly deleted by a user.

3.3.6. Content, Anchors, and Hyperlinks

3.3.6.1 Content of Nodes and Links

In addition to user-defined attributes, each node and link has a content variable which is a pointer to its logical content. The scope and behavior of content is similar to that of common attributes; both are meaningful in all graph contexts. Since content is stored externally to nodes and links, content may be stored in a file or the content may even be a whole graph. Informally, we say that a node or link contains a file or that it contains a graph. Formally, we mean that the content variable of the node or link points to a file or points to a graph.

When a node or link has content, it is called the container of the content. The relationship between content (files and subgraphs) and containers (nodes and links) is one-to-one. Thus, a node or link has at most one file or graph as its content; and each file and graph is the content of one and only one node or link. The only way for a user to create a graph or a file in the data model is to create it as the content of a particular node or link. Content is automatically destroyed when its container is destroyed.

An application can use a file which is the content of a node to store any form of data. File content can be used to store text, bitmaps, drawings, digitized audio and video, spreadsheets, or any other data. For example, an application can represent the hierarchical nature of a document by storing the document in a subgraph which is a tree. In this case, a paragraph in the document will correspond to a node in the tree. If the text of the paragraph is stored in the file which is the content of the node, then the text file will be given the same protections as the node itself. In this way, the graph model incorporates and extends the traditional concept of a file.

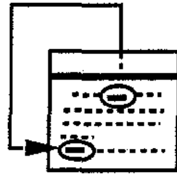
3.3.6.2. Anchors

An anchor is an application-specific pointer into the content of a node. Normally, the target of a hyperlink is an entire node. However, an anchor can be used to narrow the target of a hyperlink to a specific place within the content of a node. Each node contains a *link table* which associates hyperlinks with specific anchor points within the node's content. More specifically, a link table is composed of (hyperlink, anchor) pairs. Applications are responsible for maintaining the value of the anchor so that an anchor value always points to the same part of the node content. This responsibility must belong to the application, since in general the DGS will not have knowledge of the structure of the content.

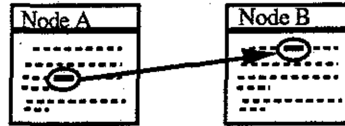
3.3.6.3. Anchored Hyperlinks

An anchored hyperlink is a hyperlink which is paired with one or more anchors in the link tables of its source or target nodes. In general, hyperlinks can be anchored in their source node, in their target node, in both, or in neither. Furthermore, a hyperlink can be paired

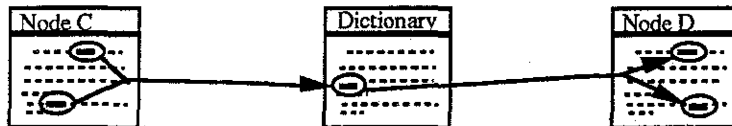
with more than one anchor within the same node. The relationship between anchors and hyperlinks is many-to-many within the context of a particular node. Several hyperlinks may be anchored to the same anchor in the node, and a single hyperlink may be anchored to more than one anchor (see Figure 9).



Reference from a place early in a chapter to a place near the end of the chapter.



Reference from a paragraph in one document to a paragraph in another document



Reference from every occurrence of a word in a paragraph to the definition of the word in a dictionary.

Reference from the definition of a word in a dictionary to every ocurrence of the word in a paragraph.

Figure 9: Examples of anchored hyperlinks.

Section 4. Synchronization of Collaborating Users

4.1. Introduction

The UNC Distributed Graph Service (DGS) will be used by groups of users as they construct a common set of hypermedia artifacts. By necessity, users will need to share and access these artifacts concurrently. The collaboration environment in general and the DGS specifically will have to provide mechanisms to make this possible. A major difficulty is the synchronization of overlapping operations by two or more users. In some cases, users will want to actively cooperate while working on an artifact. At other times, they will work independently but concurrently on the same artifact. Each of these modes of collaboration has different goals and styles of interaction. In addition, each collaborative mode places different requirements on the support environment. Some of these requirements can be satisfied at the application level, whereas others should be supported in the storage layer. In this section, we will discuss three modes of collaboration as a framework for understanding the synchronization needs of collaborating groups.

4.2. Synchronous Collaboration

Synchronous collaboration is the process by which users interactively cooperate to modify the same artifact at the same time. People are collaborating in this mode when they take turns writing on a whiteboard. An essential characteristic of synchronous collaboration is that the collaborators are interacting with each other as well as with the artifact. In many cases, the human interaction is more important than any tangible artifact which is produced. For instance, the success of a face-to-face meeting often depends on the moods and personalities of the participants. In a computer-mediated collaboration environment, all human interaction must occur within the constraints imposed by the system. To mimic face-to-face collaboration, the computer environment should provide audio/video conferencing as well as a simulated whiteboard facility. Most of this interaction occurs at the application level. Thus, although we recognize the importance of synchronous collaboration, we believe that it is best supported in a layer above the storage service.

4.3. Asynchronous Collaboration between Users whose Updates Overlap in Time but not in Space

Asynchronous collaboration is the process in which people work independently toward a common goal. Software developers do this, for instance, when they divide a programming project into modules and assign a different module to each person. If the interfaces to the modules have been well defined, then very little interaction is required between the programmers. Occasionally, two programmers may need to edit the same file, but the majority of their work can be done in isolation. Traditional file and database systems do a good job of supporting this type of collaboration in which users are working in relatively disjoint parts of the data. Since the mechanisms to support this are well-known, we do not expect that this type of asynchronous collaboration will be difficult.

4.4. Asynchronous Collaboration between Users whose Updates Overlap in both Space and Time

4.4.1. Introduction

In many cases, users need to work asynchronously on the same artifact at the same time. For instance, programmers of different modules may need to edit the same header file. If each programmer can edit the file in a few minutes, then the programmers can take turns. If, however, the changes will require hours or days to complete, then the problem is more severe. Similar problems occur when multiple readers are viewing the same artifact. When an artifact is changed, then each reader's copy of the artifact becomes out of date. If it is important for the reader to have the most recent version, then the system must provide a mechanism to support this.

What follows is a framework for describing and classifying different models of asynchronous browsing. Models of asynchronous browsing can be classified according to four orthogonal dimensions: the distance between applications, the access mode of applications, the granularity of visible updates, and the frequency of synchronization. First, we will use these dimensions to enumerate the solution space, then we will specify the particular approach that we have chosen for the DGS.

4.4.2. Distance Between Overlapping Asynchronous Applications

The distance dimension qualitatively reflects both the physical distance between applications as well as the number of layered software interfaces that must be crossed to move data from one application to another. Different distances might imply different levels of service. For example, a system might guarantee that updates which are made by an application will be immediately propagated to other applications running on the same machine. However, it might give weaker guarantees to applications running on different machines. Another natural division is between applications belonging to a single user and applications which belong to different users.

4.4.3. Access Mode of Applications

The access mode determines the type of operations that are allowed on a node, link, or graph. Read operations generally do not change information. Examples of read operations include: reading an attribute, following a link, and looking at the content of a node or link. Write operations change information. Examples of write operations include: changing the value of an attribute, creating a new node or link, removing a node or link from a graph, and changing the content of a node or link. Asynchronous browsing can be categorized according to amount of read and write concurrency that is allowed. Some useful categories include:

1. Multiple read applications for the same object.
2. Multiple read applications and
   one write application for the same object.
3. Multiple write applications for the same object.

4.4.4 Granularity of Visible Updates

The granularity of visible updates defines the size of the update which would be visible to a reader if synchronization occurred after every update. The size of an update is measured by the number of atomic DGS operations which are contained within it. Some possibilities are:

4. A writer's updates are visible after each operation.
5. A writer's updates are visible after arbitrary groups of operations.
6. A writer's updates are visible when an application closes an object.

4.4.5. Frequency with which Independent Views of the Same Object are Synchronized

The frequency of synchronization determines how often an application is synchronized to the last visible change. Some possible times that synchronization can occur are:

7. Once when an object is first opened.
8. At arbitrary times during an application.
9. Every time a visible change occurs.

The last two dimensions are confused easily since they are closely related. The frequency determines how often synchronization occurs, whereas the granularity of updates determines the point in the stream of operations to which the application is synchronized.

4.5. Synchronization in the UNC Distributed Graph Service

A perfect database would create the illusion of a central, unique copy of each object which is shared by all applications simultaneously. Unfortunately, both frequent synchronization and a very small granularity of visible updates are required to maintain the illusion. In addition, the single copy illusion places considerable requirements on the user interface to maintain displays of data which might be changing frequently. As a result, the first version of the DGS will provide a much weaker guarantee. Future versions will approximate the single copy illusion with increasing accuracy. We expect that the level of work required to provide the complete illusion is at least as large as a dissertation.

We conclude this section by classifying the DGS according to the model of asynchronous collaboration that we just defined. The four dimensions of the model were: the distance between applications, the access mode of applications, the granularity of visible updates, and the frequency of synchronization. The first version of the DGS will provide asynchronous collaboration with the following characteristics:

-- All distances will have exactly the same synchronization characteristics.
-- Multiple readers and a single writer will be supported, as will be the weaker case of multiple readers alone.
-- The granularity of updates will be that operations are visible when an application closes an object.
-- Synchronization between asynchronous applications will occur only when an object is opened.

Section 5. Operations on Graph Objects

5.1. Introduction

In previous sections we described the objects in the DGS data model, but we were purposefully vague about specific operations on those objects. In this section we will outline some specific operations for manipulating DGS data objects. Our treatment will not be exhaustive, but is intended to highlight the essential operations and their uses. A complete explanation of the operational interface will be provided in a companion document.

5.2. Concurrent Access

5.2.1. Categories of Accessible Information

Three categories of information can be accessed: system data, object data, and graph data. System data is information about an object which will be maintained by the DGS, such as creation dates, ownership, and access permissions. Object data, which will be maintained by the application, includes all data which is intrinsic to a node, link, or graph. This includes common attributes, the content of nodes and links, and the anchor table of nodes. Graph data is information about a node or link which is specific to a particular subgraph or hypergraph. Graph data includes graph attributes and intragraph linking information. In the remainder of this subsection, we will discuss concurrent access with respect to each type of information.

5.2.2. Concurrent Access to System Data

System data is sensitive information which must be protected by the DGS. Some system data, such as the creation date of an object, does not change during the life of an object. This type of system data is essentially read-only. In other cases, system data can be changed, but only by a select group of users. An example of this type of information is the access permissions for an object. All system data can be concurrently read by any authorized user. However, only one user may write to a piece of system data at a time. Thus, write requests by different users are atomic with respect to each other. When a user reads a piece of system data, the only guarantee that the DGS will make is that the data is current at the time that the read executes. Subsequent writes by other users may invalidate the reader's copy. The DGS will provide a different access operation for each type of system data. Examples include get_creation_date() and get_access_permissions().

5.2.3. Concurrent Access to Object Data and Graph Data

5.2.3.1. Operations for Accessing Object and Graph Data

Before a user can access the object data or the graph data of a database object, the user must explicitly open that object in one of two access modes: read or read_write. In addition, nodes can be opened in a third access mode called read_no_anchor. For nodes, the deletion of existing anchors is allowed in all three modes, but the creation of new anchors is allowed only in the read and read_write access modes. An open command will fail if the user lacks the proper access permissions or if the open conflicts with other opens in progress. Three open commands are provided: open link, open node, and open graph.

open link	grants access to the object data of a link
open node	grants access to the object data of a node
open graph	grants access to the object data of a graph and to all of the graph data belonging to that graph.

When a browser opens a graph in write mode, the browser receives write exclusion on all of the graph data belonging to the graph. This gives the browser permission to rearrange the links within the graph. However, to change the content of a node within the graph, the browser must open the node itself in write mode. Thus, the model encourages browsers which read structure (in graphs) and write content (in nodes) and vice-versa.

For object data and graph data, one writer only is allowed at a time, but multiple readers can exist concurrently with the writer. For the anchor data of nodes, if there is a writer, then only the writer can create anchors. If no writer exists, then a reader can open the node in create anchor mode. Multiple readers can be in create anchor mode concurrently, but a writer is not allowed to open a node that is already open by a reader in create anchor mode. Thus, one or more concurrent readers in create anchor mode can block a writer indefinitely and vice versa. These restrictions are especially suited to cases in which a particular graph or node is read-only. For example, at the beginning of the design phase, the requirements document can be thought of as read-only. An occasional user might modify such a document, but the frequent user will open it in read mode. By allowing readers to create anchors, the DGS will greatly increase the amount of concurrent access that will be possible to this type of read-only artifact.

5.2.3.2. Recovery from Application Failures

The user should execute an appropriate close command when finished with an object. The close commands--close link, close node, and close graph--are analogous to their open counterparts. At the time of a close, the user can either finalize the changes that have been made or the user can restore the state of the data that existed at the time of the open. If the application terminates before the close command has been executed, then the changes are preserved in an uncommitted state. When the application reopens an object after a failure, it can choose to resume working on the uncommitted changes or it can abort the changes and return to the last committed state. No application will be allowed to continue work on the object unless the previous changes are explicitly resumed or aborted. Depending on the operation completion semantics and the state of the application before the failure, the uncommitted version of the data may be in an inconsistent state. It is the responsibility of the application to determine the consistency of the data and either resume or abort it. Even if the changes are resumed, the application still has the option of aborting the work and returning to the last committed state.

5.2.3.3. Undo Semantics

When a writer opens an object, a copy of the object will be made and subsequent changes will be made to the original. The DGS will keep the copy as a checkpoint until the writer closes the object. The writer will be able to restore this checkpoint at any time before closing the object. When the writer closes the object, the writer's changes will become the next checkpoint and the old checkpoint will be discarded.

When a reader opens a graph, the DGS will make the reader a complete copy of the graph's latest checkpoint. The reader's copy will be internally consistent since it will reflect the last checkpointed state of the graph. However, no other guarantees will be made. For

example, it might be possible for two readers to be reading different versions of the same graph. A reader's copy of a graph will be discarded when the reader closes it.

5.2.4 Concurrent Access to Anchor Tables

To create valid anchors, an application must be viewing the most recent version of a node's content. If a reader is allowed to create anchored hyperlinks while a concurrent writer is changing the content of the same node, then the values of the new anchors might be incorrect. One way to prevent this is to restrict the number and type of users who can create anchors concurrently. We have adopted an approach which allows multiple concurrent readers to create anchors or one writer. In this model, anchor creation is a mode of access which is exclusive with respect to write access. When an application opens a node, it must specify both its read/write intent as well as its intent with regard to anchors. Thus, one or more concurrent readers in anchor creation mode will block any potential writer. Similarly, a writer will block all potential readers who want to create anchors.

We rejected an alternative approach that allowed multiple readers and a single writer to create anchors concurrently. This approach would have required read applications to maintain up-to-the-minute copies of their data. An additional problem was that it could have been defeated easily by some file-oriented applications. Consider the fact that most file-based applications cache their changes in memory and infrequently write the changes to disk. Suppose that USER A opens a node in write mode and starts writing to it using EZ edit. Also suppose that EZ edit makes all the changes in memory but does not update the file image. USER B opens the same node in read mode and creates an anchor. Even if USER B's editor gets the latest disk image, it cannot get the changes that USER A has made in memory. As a result, USER B's anchor could be invalid. This example illustrates a problem which is inherent in an open architecture which allows file-oriented applications.

5.3. Permissions and Access Control

Users can restrict each other's access to information by changing the permissions on database objects. A permission list is an access control list which identifies the categories of operations that each user or group of users can perform on a particular database object. Two permissions are defined for each node, link, and graph: administrate permission and access permission. A user with administrate permission on an object can perform administrative operations such as changing the object's permissions. Access permission is required to access the object data of nodes, links, and graphs or to access the graph data associated with a graph. When a node or link is created, the permissions of the graph are copied into the node or link. Thereafter, the permissions of the node or link are independent of the graph. A special group called "super" has all permissions on all objects in the database.

The *administrate permission* gives a user permission to change the permissions on an object, including the administrate permission itself. The administrate permission makes it possible to separate the concepts of creator and owner. The creator will always be a single user, but the owner or administrator of the object could be a different user or even a group of users. The administration of an object can be easily transferred by changing the group of users who have the administrate permission.

The *access permission* gives a user the permission to access the data associated with an object. This at least includes access to the object data, but for a graph it also includes

access to all of the graph data belonging to the graph. Two degrees of access are distinguished: read only and read/write (see Table 1).

Table 1: Access Permissions and what they provide.

Object Type	Permissions and What They Provide: read	read_write
Node	read anchors read node attributes and content	change anchor values change node attributes and content
Link	read link attributes and content	change link attributes and content create/delete anchors associated with the link
Graph	read common attributes of graph read graph attributes of nodes and links in the graph read structure of graph	change common attributes of graph change graph attributes of nodes and links in graph change structure of graph

Permission lists allow permissions to be granted to groups of users as well as to individuals. Groups of users can be identified by user-defined group names. Any user can create a user-defined group which is a list of users and user-defined groups. Only the creator of a user-defined group can change its membership. Since a user can be a member of more than one group, it is possible for a user to have two or more different sets of permissions for the same data. The complete set of permissions which are held by a user is determined by taking the transitive closure of the group membership relation. Within the complete set of permissions belonging to a single user, less restrictive permissions override more restrictive permissions. For example, read/write permission overrides read permission. A special group called "super" has all permissions on all objects in the database.

The permissions of a user are determined at the time that a node, link, or graph is opened. A user will not be allowed to open an object in any mode without the appropriate permissions. For a particular user, the permissions which are in effect at the time of an open will remain effective until the corresponding close for that object.

5.4. Anchor-Related Operations

A hyperlink connects two nodes which may or may not be in the same subgraph. Applications can perform finer-grained linking by creating anchored hyperlinks to and from anchor points within nodes. An anchor is an application-specific extension of a hyperlink into the content of a node. Although anchors should point to a specific place within the content of a node, the use and consistency of anchors is application-dependent. In most cases, applications will update the value of an anchor as the content of a node changes, since anchor values are dependent on a particular state of the content. Applications are solely responsible for updating content and for maintaining the consistency of anchor values.

Anchors are extensions to links which cannot exist in the database unless one or more links are associated with them. To create a new anchor, an application must execute either anchor_at_source() or anchor_at_target(). Similarly, applications must execute either unanchor_at_source() or unanchor_at_target() to remove an existing anchor. An anchor will be automatically deleted when the last link to it is deleted or when it is unanchored

from that link. However, anchored links will continue to exist even if their anchors are deleted. In this case, the links will simply become unanchored.

When an application creates content in a node, it must specify a type for the node's new anchor table. This type should indicate which application-dependent anchor format will be used in the anchor table. Once the content has been created, applications can retrieve it by using the get_content() call. To prevent accidental damage to the anchor table, the DGS will require applications to correctly specify the anchor table format in the get_content() call. Applications which fail to specify the correct format will not be allowed to open the content in write mode nor will they be able to create anchors in the node. However, the anchor table is primarily the responsibility of the application, since the DGS alone cannot guarantee the consistency of the anchor table. The DGS can only verify that the type given in the get_content() call is the same as the type which was specified when the content was created.

When an application opens the content of a node, it will receive a copy of the node's anchor table and a copy of the node's link table. An anchor table has entries of the form: (anchor ID, anchor value). Link tables have entries of the form: (direction, hyperlinkID, anchorID). Applications are free to modify the anchor values in their copies of the table. When an application closes a node which was opened in read or read_write access mode, it must give the (potentially modified) anchor and link tables back to the DGS as a parameter to the put_content() call. However, only read_write applications will be allowed to change the anchor values associated with pre-existing anchorIDs. To create or remove an anchor, an application must open a node in either read or read_write access mode and must have read_write permission on the link which is being anchored.

5.5. The Creation of Public and Private Views

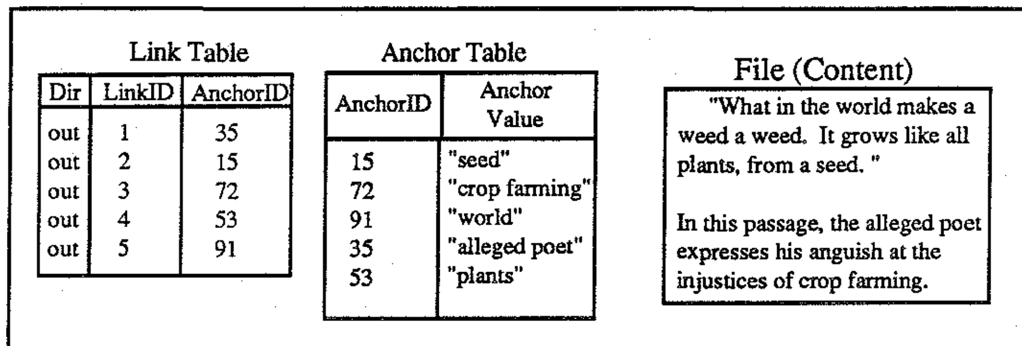
Applications are responsible for distinguishing separate personal and public views of information. Private views are created by individuals or by small groups of individuals for their own uses such as creating personal annotations. Private views should not interfere with or clutter the public view of data. In addition, subgraph owners may want to restrict the set of users who have permission to create publicly viewable hyperlinks to their subgraphs. Applications should distinguish personal and public views using the basic mechanism of grouping links into subgraphs and hypergraphs.

Public and private views are especially important with respect to anchors since a node usually has more anchors than the user wants to see. It is the responsibility of the application to determine which subset of the anchors should be displayed to a particular user. The DGS data model provides hypergraphs as the primary mechanism for distinguishing public anchors from private anchors. Applications should group public hyperlinks into hypergraphs which are publicly accessible and private hyperlinks into private hypergraphs. Before an application can determine the set of anchors to display, it must first enumerate the set of hypergraphs that the user is interested in. Next, the application should collect the set of hyperlinks which emanate from the node within the contexts of the chosen hypergraphs. Then, the application should map this set of hyperlinks to a set of anchorIDs in the node. The link table is used to perform this mapping. The result is the subset of the node's anchors that the user wants to see.

For example, a user is often interested in anchors which are a part of the public view but is not interested in private anchors that other users have created. Figure 10 shows the internal data structures of a node which contains a text file and has five anchors. The link

table and anchor table are used to find the set of anchors which are associated with a hyperlink. For example, two steps are needed to find the anchor which is associated with Hyperlink 1. First, the link table is searched for the LinkID 1. Node Z's link table has the entry (1, 35) which signifies that Hyperlink 1 is associated with AnchorID 35. Then, the anchorID is looked up in the anchor table. Node Z's anchor table has the entry (35, "alleged poet") which indicates that AnchorID 35 is associated with the Anchor Value "alleged poet". Thus, Hyperlink 1 is anchored to the phrase "alleged poet". When applications open a node, they receive the full link table and the full anchor table. In most cases, however, the user does not want to see all of the anchors.

Data Structures within Node Z



Link Table

Dir	LinkID	AnchorID
out	1	35
out	2	15
out	3	72
out	4	53
out	5	91

Anchor Table

AnchorID	Anchor Value
15	"seed"
72	"crop farming"
91	"world"
35	"alleged poet"
53	"plants"

File (Content)

"What in the world makes a weed a weed. It grows like all plants, from a seed. "

In this passage, the alleged poet expresses his anguish at the injustices of crop farming.

Figure 10: A node with a text file as its content and five anchors.

In this example, we will assume that three groups of users can access this node. Two of the groups represent individual users: User Alpha and User Beta. In addition, User Alpha and User Beta belong to a public group called the R&D Department. Figure 11 shows the five hyperlinks (1 through 5) which are represented in the left column of the node's link table in Figure 10. Hyperlinks 1 and 2 are public hyperlinks which belong to the R&D Department. Hyperlink 3 is a private hyperlink which belongs to User Alpha; and hyperlinks 4 and 5 belong to User Beta.

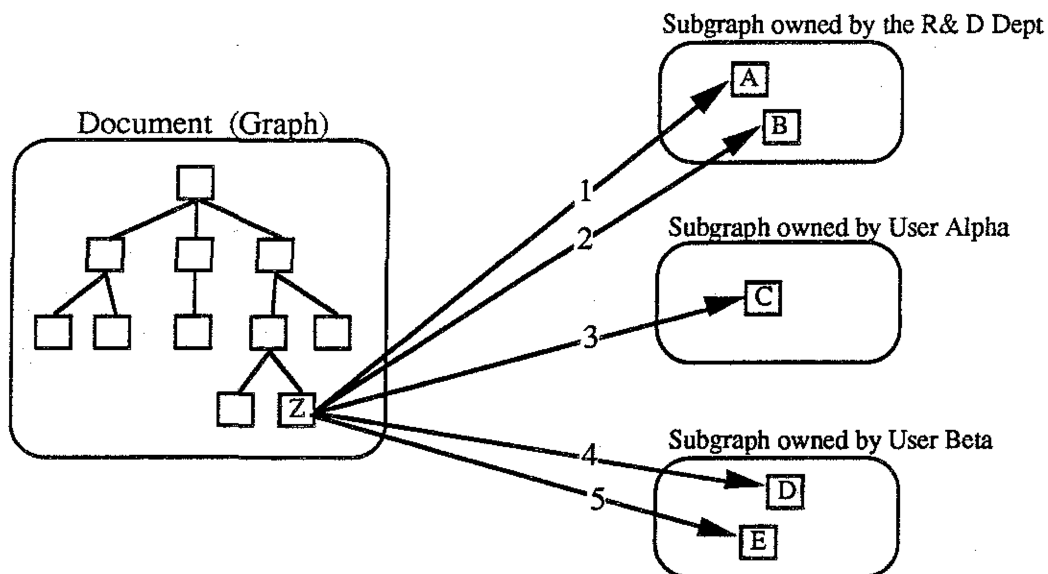
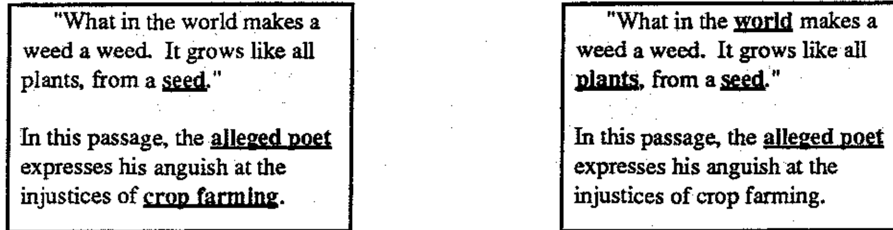


Figure 11: The five hyperlinks which emanate from Node Z.

Normally, when User Alpha opens Node Z, the user expects to see the R&D anchors but does not want to see the anchors which have been created by User Beta. Similarly, User Beta wants to see her own anchors and the public anchors, but does not want to see User Alpha's anchors. Figure 12 illustrates the views that the two users expect to see when they open the node.



The view that User Alpha wants to see. The view that User Beta wants to see.

Figure 12: Each user expects to see a different set of anchors when the node is displayed.

The problem which confronts an application is how to determine the set of anchors to display. The DGS data model provides the hypergraph as the basic mechanism to make this decision. By some application-level convention, applications must agree to group hyperlinks into hypergraphs which correspond to user views. In this simple example, applications might have agreed to group the hyperlinks into three hypergraphs (see Figure 13): one corresponding to the R&D Department (public), one corresponding to User Alpha, and one corresponding to User Beta. Given these hypergraphs, an application could determine User Alpha's view by unioning the hyperlinks in the public hypergraph with the hyperlinks in User Alpha's hypergraph. This would result in the set of hyperlinks {1, 2, 3}. Next, the application could look in Node Z's link table to find the corresponding anchor IDs. This would result in the set of anchor IDs {35,15,72 }. Finally, the application could look in Node Z's anchor table to find the corresponding anchor values {"seed", "alleged poet", "crop farming"}. Through a similar process, an application could determine that User Beta is interested in the anchors {"seed", "alleged poet", "world", "plants"}.

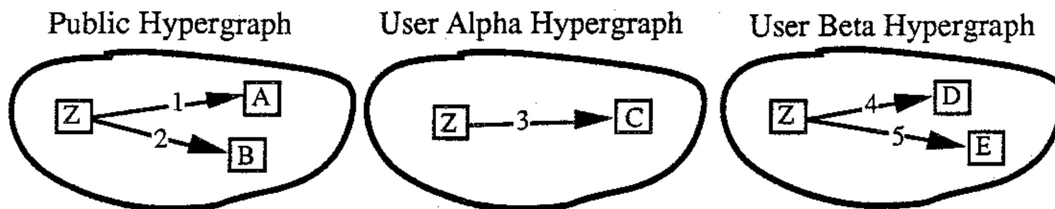


Figure 13: The five hyperlinks could be grouped into three hypergraphs.

This example illustrates a basic technique that applications can use to distinguish public from private views of information. However, real situations are more complicated and will require applications to use hypergraphs in more sophisticated ways. Nevertheless, the example makes several important points. The example shows that different users may legitimately desire different views of the same data. The hypergraph is shown to be an effective mechanism for distinguishing these views. For this to work, different applications must agree on a convention for using hypergraphs. In the example, we suggested grouping hyperlinks according to the groups of users who might want to view

them. The reader should note, however, that this is just one possible convention. Real applications are likely to use hypergraphs in more complicated ways and, thus, will require more complicated conventions.

5.6. Operation Completion Semantics

Operation completion will be ambiguous in the case of failure, since some requests may be buffered at certain places in the system. However, an explicit sync or flush command will be provided to establish checkpoints. When a sync command completes, the DGS will guarantee that all of an application's operations prior to the sync have been completed and that they will not be affected by future crashes. Furthermore, an automatic sync mechanism will periodically do a background sync on behalf of the user. The frequency of automatic syncs will be controlled by the user through a few tunable parameters such as the maximum number of seconds between syncs and the maximum number of unflushed operations.

Section 6. References

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