Implementation Design Document
for the
UNC Distributed Graph Service
TR92-031
March, 1992

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

UNC is an Equal Opportunity/Affirmative Action Institution.
Abstract

We are designing a distributed hypermedia storage service—called the UNC Distributed Graph Storage System (DGS)—which will become the database for a computer-supported cooperative work environment called ABC. The DGS data model has been designed to support the storage needs of users who are collaborating to solve complex problems. This document describes the implementation of the DGS.

Acknowledgements

This work was done mostly during the Summer and Fall 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 Shankar Krishnan, John Menges, Michael P. Wagner, and Zhenxin Wang. 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. Overview ........................................................................................................1
  1.1. Partitions ...............................................................................................................1
  1.2. DGS Processes .......................................................................................................5
    1.2.1. Application Processes ....................................................................................6
    1.2.2. Graph Server Processes ..................................................................................7
    1.2.3. Storage Server Processes ................................................................................9
Section 2. Detailed Examples ........................................................................................12
  2.1. Establishing a Connection with the DGS and Modifying a Subgraph ..................12
  2.2. Anchor Table Merging ..........................................................................................13
  2.3. An Example Involving Hyperlinking and Content ...............................................16
    2.3.1. Open a pre-existing tree and create a hyperlink from one of its nodes to another.................................................................16
    2.3.2. Create a new tree and create a hyperlink from a node in the new tree to a node in the old tree .........................................................18
    2.3.3. Edit the Content of the Root Node of Tree T2 ..............................................21
Section 1. Overview

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 Storage System (DGS). In this report, we describe the design of the DGS and the implementation that is in process.

1.1. Partitions

The Distributed Graph Storage System (DGS) provides applications with the illusion that it is running on a single machine and is composed of a flat universe of nodes, links, and graphs. In reality, the graph store is divided into non-overlapping collections of nodes, links, and graphs called partitions which are distributed and replicated by the DGS. Partitions form the boundaries for administrative controls such as space quotas and replication policies. Each database object is completely contained within a partition. By default, nodes and links are created in the same partition as the graph in which they are created. However, this does not prevent nodes and links from being shared by graphs which are stored in different partitions. Graphs which are created as the content of a node or link, are placed in the same partition as one of the graphs which contains the node or link. For example, if an application creates Subgraph C as the content of Node B while it is viewing Subgraph A, then the DGS will create Subgraph C in the same partition as Subgraph A. In most cases this results in the node and its content being stored in the same partition, although this is not always true. For instance, if Node B and Subgraph A were contained in different partitions, then Subgraph C would not be created in the same partition as Node B.

A typical use for partitions will be to allocate space for a new user. First, a system administrator will create a new partition and allocate space to that partition according to the needs of the user. Next, the new user will be given read-write permission on a subgraph which the administrator creates in the new partition. This subgraph will be called the user's home graph and will serve a function similar to a user's home directory in UNIX. Finally, the home graph will be attached to the root subgraph.

When a user has read_write permission on a graph, then the user can create nodes in the graph and, as a consequence, in the partition which contains the graph. A user cannot create objects in a partition if the user does not have read_write permission on at least one graph in the partition. Thus, a user's access to a partition is completely determined by the user's permissions on the graphs that it contains. To illustrate, Figure 2 shows the home graph of a user named "shackelf" and the DGS database which contains it. In addition, information about the allocation of partitions is provided in Tables 1 and 2. In this example, User
Shackelf has been allocated Partition 2 which has 10 MB of space. Assuming that Shackelf has not been given read_write permission on any graphs except SG 1, then he is limited to this 10 MB allocation. This results from the fact that he can only create nodes in his home graph and that his home graph is contained in Partition 2.

Since a partition is associated with a piece of physical storage, the partition of an object is closely related to the physical location of that object within the distributed graph database. Thus, partitions also play a central role in the process of locating objects on disk. We define a logical partition to be a directory which maps Object IDs (OIDs) to the physical locations of the objects with those OIDs. The Logical Partition Number (LPN) of an object is embedded in its OID (see Figure 3). Thus, one can determine the logical partition of an object simply by examining its OID. Unfortunately, this also means that an object must remain in the same logical partition for its entire lifetime, since the OID of an object cannot be changed. It is for this reason that we distinguish the logical partition of an object from its absolute physical partition.

A physical partition is defined to be a collection of objects which are stored physically together on the same disk. While logical partitions are the basis for locating objects in the DGS, physical partitions are associated with basic administrative functions such as space quotas and the replication of user data.
Figure 3: The Logical Partition Number is embedded in the OID of an object.

The distinction between logical and physical partitions is most useful when system administrators need to move data from one physical location to another. Figure 4 shows a DGS database which contains private data belonging to User Shackelf and group data belonging to the Colab Project. The database consists of the three logical partitions which are shown in Figure 5. Each logical partition is an object location directory which maps an OID to a physical partition and a disk location. There are four physical partitions: PP 0, PP 1, PP 2, and PP 3 (see Table 3). Each physical partition is defined by three features: a physical site location, a size limit in bytes, and a replication factor which indicates the number of identical physical copies of the partition that should be maintained by the DGS. Physical partitions are replicated to improve their availability and recoverability in the event of media and process failures. For example, PP 0 is replicated three times because the root subgraph must be highly available.

For the example, we will assume that User Shackelf has just moved from the Computer Science department at UNC to a position in the CS department at Duke, although he plans to continue working closely with his colleagues in the UNC Colab project. Since he is moving to Duke, the user naturally wants to take his graphs in the DGS with him. The system administrators at UNC are happy with this because they will be able to transfer the administrative burden and space requirements for those graphs to the user’s new site. One way to accomplish this transfer would be to copy the graphs to Duke and then delete them at UNC. However, this method would have the unfortunate effect that all of the copied objects would be reassigned new OIDs. The result of this would be that any hyperlinks between Shackelf’s work and other objects at UNC would be destroyed. Clearly, this would be unacceptable since Shackelf expects to continue his previous collaboration with the UNC Colab group.
Figure 4: A More Detailed Example of a DGS Database

Figure 5: The Logical Partitions Which Are Associated with Figure 4

Table 3: The Physical Partitions Which Are Referenced in Figure 5
Fortunately, this is precisely the problem that the logical partitions were created to solve. In contrast to copying objects, the logical partition allows us to literally move objects by updating the location information in the logical partition directory. This solution is especially attractive because it preserves all hyperlinking and content relationships between the transferred objects and the rest of the database. Figure 6 shows the state of the logical partitions after SG 1, SG 2, SG 3, SG 4, SG 5, and SG 8 have been moved from physical partitions at UNC to a physical partition at Duke. As shown, the updated version of Logical Partition 2 now contains some objects that are stored at UNC in PP2 as well as an object that is stored at Duke in PP3. In addition, Logical Partition 1 also contains objects that are stored in PP3. Thus, the correspondence between logical partitions and physical partitions is many-to-many.

<table>
<thead>
<tr>
<th>Logical Partition 1</th>
<th>Logical Partition 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID</td>
<td>Physical Partition #</td>
</tr>
<tr>
<td>SG 1</td>
<td>PP 3</td>
</tr>
<tr>
<td>SG 2</td>
<td>PP 3</td>
</tr>
<tr>
<td>SG 3</td>
<td>PP 3</td>
</tr>
</tbody>
</table>

Figure 6: The Logical Partitions after the Graphs have been Moved to Duke

In this section we introduced two central concepts: the logical partition and the physical partition. Each object in the database is contained in exactly one logical and physical partition. Its logical partition, which remains constant for the life of the object, is used to locate the object in the distributed store. In contrast, the physical partition of an object can change as the object is moved from one location to another and is the basis for administrative controls such as space quotas and data replication. Despite their importance in the daily use and maintenance of the DGS, partitions are invisible to average users. Access permissions are associated with graphs rather than with partitions. Thus, users view the graph as the central tool for grouping, allocating, and restricting space. The DGS has been carefully designed so that only system administrators and system programmers need to understand partitions. In the next section, we will introduce the basic process architecture which provides this support.

1.2. DGS Processes

The DGS is composed of five types of processes: applications, graph servers, storage servers, protection servers, and matchmakers (see Figure 7). Storage servers are responsible for storing partitions on disk.
replicating partitions for availability and fault tolerance, and recovering from failures. Each graph server communicates with multiple storage servers to hide the existence of partitions and provide the illusion of a seamless graph store. Each host machine runs a single graph server process which services all applications which are running on that machine. Protection servers maintain a database of group membership information. Matchmaker processes facilitate connection establishment between applications and graph servers, and between graph servers and storage servers.

![Diagram of DGS Processes]

Figure 7: The Five Types of DGS Processes

1.2.1. Application Processes

An application process acts on the behalf of a user to read and modify objects in graph storage. An application program will be composed of two layers: an application layer and a graph client layer (see Figure 8). The graph client layer will export to the application layer a set of graph-oriented functions and will handle all of the communication with DGS processes. Normally, the graph client will communicate only with a single graph server process on its machine, although in times of initialization and recovery it might also communicate with processes on other machines. RPC protocols will be used for interprocess communication between graph clients and graph server processes. Although ISIS will be used extensively in the implementation of the DGS, application processes will not need to use any ISIS services directly.
1.2.2. Graph Server Processes

A graph server process performs the graph-oriented operations that one or more applications request through their graph client interfaces (see Figure 9). The graph server is essentially an intelligent cache which maintains nodes, links, and graphs in its local memory as well as copies of file content on its local disk. Since graph servers do not permanently store data, they depend on special storage server processes to do this.

Figure 8: The Two Layers of an Application Process

Figure 9: Graph servers implement the graph data model and provide transparent access to multiple partitions.
The graph server process is responsible for implementing all graph operations except for anchor table merging. When an application requests that an object be opened, the graph server process retrieves the object from a storage server using a simple block-oriented protocol. As the application makes requests, the graph server will perform those requests on its local copy of the object. However, none of those operations will be reflected in the storage server until the graph server converts the modified object into a block of bytes and then stores that block in the storage server. Generally, the graph server will only do this when the application explicitly requests that the object be closed. Thus, the graph server must log enough information to recover the application’s work in the event that it crashes before that work has been stored safely. When recovering after a failure, the most information that a graph server can expect to retrieve from a storage server is the initial state of any objects that were open when it failed. Thus, the graph server is responsible for logging all write operations which occur between the time that an object is opened and the time that it is closed.

Based on the operation completion semantics that are outlined in the requirements document, a particular application may select a logging policy that trades risk for speed. As a result, the log may not be flushed to disk after every write operation. In actuality, the graph server will flush its log when N graph operations have been executed by an application or when M seconds have passed or when the application performs an explicit flush operation. N and M can be set independently by each application. A zero value in either N or M corresponds to the situation in which the log is flushed on every operation.

Graph servers are responsible for maintaining the consistency of graphs and the structure of typed graphs. For example, consider the following list of operations to a subgraph:

- open subgraph
- create node in subgraph
- create common attribute in node
- close and save node
- close and abort subgraph

In this example, the application has made two contradictory requests: saving the node and aborting the changes to the subgraph. Clearly, if the changes to the subgraph are aborted, then the creation of the node must also be aborted. It is the responsibility of the graph server to enforce reasonable graph semantics. A similar problem can occur between links and anchors if an application creates a link, anchors the link, and then aborts the link. The graph server must abort both the link and any anchors which were associated with it. This is representative of a whole class of graph consistency problems which the graph server must detect and correct.
Each graph server contains a cache manager which preloads data for a client based on the concept of locality of reference. We assume that applications will make queries on a very small scale (many operations are at the granularity of individual attributes). If a client asks for attribute x of Node A, it will probably ask for attribute y of Node A very soon. So the graph server will preload all attributes of Node A for future reference. This preloading of data is accomplished by communication with storage server processes. At least two degrees of preloading are possible: brief summary and whole object. Summary information refers to the common and system attributes of an object, but does not include things such as the content of nodes or the structural information of graphs. Ideally, one could always load the whole object; however, the reality is that memory and network bandwidth are scarce resources. When a user is evaluating a large number of potential paths, it is often better to limit the initial accesses to summary information and load the detailed content of the object only when it is requested. Both the graph server and the storage server will provide mechanisms for transferring both summaries and whole objects. Applications will be provided with a mechanism which allows them to provide hints to the graph server about the amount of detail that is required when an object is opened. Similarly, operations in the interface to the storage server will allow the graph server to request either the whole object or the summary alone. However, at all times it is the graph server which makes the final decision about how much data is needed. The graph server is responsible for taking hints from the application and then making reasonable decisions about how much information should be cached.

Each graph server will be contained in its own ISIS process group in order to benefit from the broadcast, logging, and transaction management capabilities of ISIS.

1.2.3. Storage Server Processes

Storage server processes are responsible for the permanent storage of data on disk (see Figure 10). Each server maintains one or more physical partitions. The interface to the storage server is designed to isolate the graph servers from the details of physical storage. The primary responsibility of the storage server is to store and control access to uninterpreted blocks of bytes, which are indexed by a unique Object IDentifier (OID). However, storage servers also provide a semi-centralized service for creating OIDs and anchor IDs within the partitions that they maintain. In addition, they also merge anchor table information which is created by concurrent readers of the same node.

Storage servers allocate Object IDentifiers (OIDs) and anchor IDs to graph servers in blocks of 256 and 16, respectively. Each block of OIDs represents a valid range of OIDs for objects within a particular partition. Thus, graph servers can create up to 256 new objects in a partition between communications with a storage
server. A block of anchor IDs represents a valid range of anchor IDs within a particular node. Graph servers allocate identifiers to applications in response to object and anchor creation requests.

Figure 10: Storage servers provide multi-site replication of data and permanent disk storage.

Users can limit each other's access to objects in the DGS by changing the permissions that are associated with the objects that they administrate. No user is allowed to access an object unless the user has the proper permissions for the mode of access that is requested. Storage servers are responsible for enforcing and permanently maintaining permission information. Although graph servers may keep read-only copies of permissions, updates which are not reflected in a storage server are not permanent nor are they globally visible. When a graph server requests an object on behalf of a user, the storage server must first check the permissions of the user and then check to see if the access is in conflict with the activities of other users. If there is no problem, then the storage server will send a copy of the object along with its permissions back to the requesting graph server. The permission information which will be sent to the graph server will include a list of groups and their respective permissions, but it will not include any group membership information. Permission information will be sent to the graph server so that read-only requests for
permission information can be satisfied locally to the application. An example in which this might provide a significant performance improvement is when a graph is used to represent a UNIX-style directory.

All changes to permission information must go through the storage server. Thus, even if a graph server has a copy of an object's permissions, it must explicitly notify the storage server whenever it desires to change the object's permissions. Thus, a typical scenario is this:

application requests object
  graph server requests object
  storage server sends object and permissions to graph server
application requests permission information
  graph server retrieves permission information from its cache
application requests that the permissions on the object be changed
  graph server forwards request to storage server
  storage server sends an updated version of the permissions to the graph server
  graph server returns either success or failure to application

In almost all cases, storage servers are solely responsible for enforcing the semantics of permissions. Usually, graph servers do nothing more than read-cache the permission information. However, the creation of anchors is a notable exception. In order for an application to anchor a link, the user of the application must have write permission on the link. In this case only, it is the responsibility of the graph server to make sure that the user has the appropriate permission. Thus, the graph server will have to ask the storage server for a list of the permissions which the user has on the link that is being anchored. Based on this information, the graph server will then determine whether to accept or deny the anchoring request. This does not require that the graph server send a query the protection server directly. If there is a protection server which is separate from the storage server, then the storage server should query the protection server on behalf of the graph server. It is more reasonable for the storage server to interact with the protection server because the storage server is probably physically closer to it than the graph server.
Section 2. Detailed Examples

In the previous section we provided an overview of basic components of the DGS design and implementation. Those components included logical and physical partitions, applications, graph servers, storage servers, protection servers and matchmaker processes. In this section we will focus on the interaction among those components as the DGS performs a typical set of tasks. We will begin by outlining the process by which a new application connects to the DGS. The section will conclude with a detailed example in which a user creates a hyperlink.

2.1. Establishing a Connection with the DGS and Modifying a Subgraph

1. Application executes DGS_Connect() which is defined in the interface which is exported by the graph client layer. The graph client layer looks up the address of the nearest matchmaker using a name service.

2. Graph client sends a Connect request to the matchmaker and an authentication exchange occurs.

3. The matchmaker creates a graph server process on the same host as the client, if one does not already exist. Then, the matchmaker sends the server’s host and port number to the client.

4. The graph client layer contacts the graph server. The graph server starts monitoring the application process. The client layer sends an open subgraph request to the server.

5. The graph server extracts the logical partition number from the OID of the subgraph and contacts a process which maintains that logical partition. From the logical partition, the graph server receives sufficient information to contact the storage server which is maintaining the physical partition which contains the subgraph. Note: Depending on the specific implementation, a storage server may maintain both the logical and physical partition information for an object. In that case, contacting the logical partition would be the same as contacting the storage server.

6. Once in contact with the storage server, the graph server sends an open request for the object with the subgraph’s OID. The storage server must perform several types of checking before it is able to satisfy the request. First, the storage server must determine whether the user who is running the application has the correct permissions to open the object in the requested access mode (read or read_write). Then, the storage server must determine whether the request is in conflict with any other user requests. For example, a reader is not allowed to open a node in read-with-anchor mode if another user already has the node open in read_write mode. If these checks are passed, then the storage server sends the object to the graph server. Since the storage server stores blocks of bytes, rather than objects, the graph server must convert the bytes to an object before it can use it.

7. The graph server converts the block of bytes to a data structure that it can manipulate in memory and logs the fact that it has opened the object.

8. The graph server notifies the application that the open was successful.

9. As graph operations are received by the graph server, they are applied to the local copy of the subgraph and logged to disk. The graph server flushes its write log when an application closes an object or executes an explicit flush operation. In addition, flushing may occur automatically when buffers become full or when timeouts occur.
10. When the application finally sends a close subgraph request to the graph server, the server encodes the subgraph’s data structure into a block of bytes and sends the block to the storage server as a part of a close request. The storage server receives the block, writes it to disk, and updates its access control tables to indicate that the object has been closed. Just before the close operation completes, the graph server logs a close record to its write log.

### 2.2. Anchor Table Merging

Because the DGS allows multiple readers to create anchors in the same node at the same time, the DGS must be able to merge two or more versions of a node’s anchor table. To illustrate, this section will present an example in which two applications open the same node and then simultaneously modify its anchors. Figure 11 shows the contents of Node A when the two applications first open it in read mode. The content, link table, and anchor table of the node are transferred from a storage server to the graph servers of the two applications and then to the applications themselves.

<table>
<thead>
<tr>
<th>Link Table</th>
<th>Anchor Table</th>
<th>File (Content)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dir</td>
<td>LinkID</td>
<td>AnchorID</td>
</tr>
<tr>
<td>out</td>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>out</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>out</td>
<td>3</td>
<td>72</td>
</tr>
<tr>
<td>out</td>
<td>4</td>
<td>53</td>
</tr>
<tr>
<td>out</td>
<td>5</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| | | | | | "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 11: The initial contents of Node A.

Application 1 executes three operations: Unanchor( Link 1, Anchor 35 ), Unanchor( Link 3, Anchor 72 ), and Anchor_at_Source( Link 9 ). As a result of the two Unanchor() operations, Graph Server 1 marks the delete bit of the first and third entries of Node A’s link table (see Figure 12). When the graph server receives the Anchor_at_Source() command, it allocates a new anchor ID (104), and adds the new anchor to its copy of the link table. The application receives the new anchor ID and adds the anchor to its own copy of the tables. At this point, the user has the option to save or discard the work. If the user decides to abort the work on the node, then the graph server can simply discard the modified link table. If, however, the user decides to save the node, then the application must transfer the anchor table back to graph server (as a part of the put_content() operation). It is not necessary to transfer the link table. Then, when the graph server receives the Close() command from the application, it sends the link and anchor tables to the storage server. After that, the changes are permanent and will be seen by any user who subsequently opens the node.
Figure 12: Application 1 opens Node A, unanchors two links, and anchors one link.

Figure 13 shows Node A's tables after the storage server has received the changes from Graph Server 1. Notice that the storage server has not deleted any of the anchors. Instead, it has marked the first and third entries of the link table. The marked entries will be deleted when the last application closes the node. The reason that they are not deleted immediately is that some of the information is necessary to merge the deletions with other versions of the tables. Marked entries in the link table are never propagated outside of the storage server. Thus, if a new application were to open the node shown in Figure 13, it would only see the entries associated with the links 2, 4, 5, and 9.

Figure 13: State of Node A in the storage server after Application 1 closes the node.

At the same time that Application 1 is reading the node, Application 2 opens it as well. Application 2 executes three operations: Unanchor(Link 2, Anchor 15), Anchor_at_Target(Link 14), and Anchor_at_Source(Link 11, Anchor 35). As a result of the Unanchor() operation, Graph Server 2 marks the delete bit of the second entry in its copy of Node A's link table (see Figure 14). When the graph server receives the Anchor_at_Target() command, it allocates a new anchor ID (103), and adds the new anchor to its copy of the link table. The application receives the new anchor ID and adds the anchor to its own copy.
of the tables. When the user decides to save the node, then the application transfers the anchor table back to graph server (as a part of the put_contentO operation). It is not necessary to transfer the link table. Then, when the graph server receives the Close() command from the application, it sends the link and anchor tables to the storage server. At this point, the changes are permanent and will be seen by any user who subsequently opens the node.

<table>
<thead>
<tr>
<th>Link Table</th>
<th>Anchor Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Del</td>
<td>Dir</td>
</tr>
<tr>
<td>out</td>
<td>1</td>
</tr>
<tr>
<td>out</td>
<td>2</td>
</tr>
<tr>
<td>out</td>
<td>3</td>
</tr>
<tr>
<td>out</td>
<td>4</td>
</tr>
<tr>
<td>out</td>
<td>5</td>
</tr>
<tr>
<td>in</td>
<td>14</td>
</tr>
<tr>
<td>out</td>
<td>11</td>
</tr>
</tbody>
</table>

Graph Server 2

Application 2

Figure 14: Application 2 opens Node A before Application 1 closes it. Application 2 unanchors one link and anchors two links.

When the storage server receives Application 2's version of the tables, it must merge them with the tables that were received from Application 1. To accomplish the merge, the storage server logically concatenates the two versions and then eliminates duplicates. If two entries exist for the same anchor and one of them has the delete bit set, then that entry is kept. Thus, an entry which is deleted by any user will remain deleted when it is merged with the tables of other users who did not delete it. When the last application has closed the node, the storage server can discard any entries in the link table which have the delete bit set and any entries in the anchor table which are not referenced by an undeleted entry in the link table. Figure 15 shows the final state of the tables after both applications have closed.

<table>
<thead>
<tr>
<th>Link Table</th>
<th>Anchor Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Del</td>
<td>Dir</td>
</tr>
<tr>
<td>in</td>
<td>14</td>
</tr>
<tr>
<td>out</td>
<td>11</td>
</tr>
<tr>
<td>out</td>
<td>4</td>
</tr>
<tr>
<td>out</td>
<td>5</td>
</tr>
<tr>
<td>out</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 15: State of Node A in the storage server after Application 2 closes the node.
2.3. An Example Involving Hyperlinking and Content

In the following example a user will open a pre-existing tree (T1) and create a hyperlink between two nodes in the tree. Then, the user will create a second tree (T2) and create a hyperlink from a node in the second tree to a node in the first tree. Figure 16 shows the basic structures which will be manipulated in the example. The example assumes that the application associates a hypergraph with each subgraph. The purpose of the associated hypergraph is to store all of the hyperlinks which are coming to or from nodes in the subgraph. A second assumption is that subgraphs are created as the content of a node.

![Diagram of hypergraph structures](image)

Figure 16: The structures which participate in the hyperlinking example.

DGS_Connect()

Graph client contacts the nearest matchmaker. Authentication occurs. Graph client connects with the local graph server process.

2.3.1. Open a pre-existing tree and create a hyperlink from one of its nodes to another.

Open_Subgraph( T1, read ); /* open tree T1 for read */

The graph client sends an open subgraph request to the graph server. The graph server extracts the logical partition number from the subgraph’s OID and contacts a process which maintains that logical partition. From the logical partition, the graph server receives sufficient information to contact the storage server which is maintaining the physical partition which contains the subgraph. Next, the graph server sends an open subgraph request to the appropriate storage server. Before processing the request, the storage server verifies that the user has the proper permissions to access the subgraph and that the request does not conflict with other user requests in progress. Since the permissions are checked in the storage server, the
permission data does not have to be transferred to the graph server. If the permissions are ok, the storage server makes a copy of the last permanent version of the subgraph and then sends the copy back to the graph server. The storage server keeps this copy until the subgraph is closed. The copy is used to satisfy subsequent read requests from the graph server. Because the copy is static, the graph server can re-read the subgraph as many times as it needs and will get the same version from the storage server every time. This feature makes it possible for the graph server to discard information in its read cache. Since the storage server stores blocks of bytes, rather than objects, the graph server must convert the bytes to an object before it can use it.

Root( T1 ); /* returns the OID of the root node of T1 */

Graph server looks up the information in its local cache and returns the answer N1.

Children( N1, T1 ); /* returns the OIDs of the children of N1 in T1 */

Graph server looks up the information in its local cache and returns the answer N2 and N3.

Get_Attribute( T1, 'HGRAPH' ); /* returns the value of T1's common attribute 'HGRAPH'. For example, the application might use this attribute to store the OID of a hypergraph. */

Graph server looks up the information in its local cache and returns the answer HG1. In this example, the application uses hypergraph HG1 to store the hyperlinks which come to nodes in T1.

Open_Hypergraph( HG1, write ); /* open hypergraph HG1 for write. The application is preparing to create a hyperlink */

The graph server extracts the logical partition number from the HG1 OID and contacts a process which maintains that logical partition. From the logical partition, the graph server receives sufficient information to contact the storage server which is maintaining the physical partition which contains the hypergraph. Next, the graph server sends an open hypergraph request to the appropriate storage server. Before processing the request, the storage server verifies that the user has the proper permissions to access the hypergraph and that the request does not conflict with other requests in progress. Since the permissions are checked in the storage server, the permission data does not have to be transferred to the graph server. If the permissions are ok, the storage server makes a copy of the last permanent version of the hypergraph and then sends the copy back to the graph server. The storage server keeps the copy and the original until the user closes the hypergraph and either aborts her changes or replaces the original version. While HG1 is open for writing, all Open_Hypergraph( HG1, read ) requests will receive a copy of the original version. In addition to performing read and write operations on its local caches, the graph server also logs write operations to disk. Since the storage server stores blocks of bytes, rather than objects, the graph server must convert the bytes to an object before it can use it.

Because the graph was opened for writing, the storage server allocates a block of OIDs to the graph server for the sole purpose of creating new objects in HG1. The storage server depends on the graph server to use the OIDs for their intended purpose.

Create_Hyperlink( N2, N3, HG1 ); /* Create a hyperlink from node N2 to N3 in HG1 */

The graph server creates the new link using an OID from the pre-allocated block of OIDs that it received when it opened HG1 in write mode. Since the graph server has already been allocated a block of OIDs, it does not have to communicate with the storage server unless it wants to. By default, the hyperlink is created in the same logical and physical partition as HG1. The graph server creates the link in its local data.
structures and returns the new OID to the client. The graph server logs the creation operation to its write log. However, the write log is not necessarily flushed to disk. The graph server buffers log entries so that it doesn't have to write to disk on every operation. The write log is flushed periodically and also when an explicit flush or close request is received from the application.

Close_Hypergraph( HG1, REPLACE ); /* Close hypergraph HG1. Flush all buffers and make all changes permanent. The REPLACE parameter indicates that the changed copy should replace the original */

The hypergraph is closed. The server encodes the hypergraph's data structure into a block of bytes and sends the block to the storage server as a part of a close request. The storage server receives the block, writes it to disk, and updates its access control tables to indicate that the object has been closed. Just before the close operation completes, the graph server logs a close record to its write log and flushes it. Since the user has decided to replace the original hypergraph, the storage server can discard the original version. All future opens on this hypergraph will receive a copy of the new version. If the graph server has not notified the storage server of the creation of the new link, then it should probably do so at this time.

2.3.2. Create a new tree and create a hyperlink from a node in the new tree to a node in the old tree

Open_Subgraph( HomeGraph, write ); /* opens the subgraph with OID HomeGraph for writing. I'm assuming that HomeGraph is a user's home subgraph. */

The graph server extracts the logical partition number from the HomeGraph OID and contacts a process which maintains that logical partition. From the logical partition, the graph server receives sufficient information to contact the storage server which is maintaining the physical partition which contains the hypergraph. Next, the graph server sends an open hypergraph request to the appropriate storage server. Before processing the request, the storage server verifies that the user has the proper permissions to access the hypergraph and that the request does not conflict with other requests in progress. Since the permissions are checked in the storage server, the permission data does not have to be transferred to the graph server. If the permissions are ok, the storage server makes a copy of the last permanent version of the subgraph and then sends the copy back to the graph server. The storage server keeps the copy and the original until the user closes the subgraph and either aborts her changes or replaces the original version. While HomeGraph is open for writing, all Open_Subgraph( HomeGraph, read ) requests will receive a copy of the original version. In addition to performing read and write operations on its local caches, the graph server also logs write operations to disk. Since the storage server stores blocks of bytes, rather than objects, the graph server must convert the bytes to an object before it can use it. Because the graph was opened for writing, the storage server allocates a block of OIDs to the graph server for the sole purpose of creating new objects in HomeGraph. The storage server depends on the graph server to use the OIDs for their intended purpose.

SomeNode := Create_Node( HomeGraph ); /* Creates a new node in the subgraph. */

The graph server creates the new node using an OID from the pre-allocated block of OIDs that it received when it opened HomeGraph in write mode. Since the graph server has already been allocated a block of OIDs, it does not have to communicate with the storage server unless it wants to. By default, the node is created in the same partition as the HomeGraph. The graph server creates the node in its local data structures and returns the new OID to the client. The graph server logs the create_node operation to its write log. However, the write log is not necessarily flushed to disk. The graph server buffers log entries so that it doesn't have to write to disk on every operation. The write log is flushed periodically and also when an explicit flush or close request is received from the application.
T2 := Create_Subgraph_As_Content( SomeNode, HomeGraph, TREE ); /* creates a new subgraph of type TREE as the content of the given node */

In this case, the graph server creates the new subgraph using an OID from the pre-allocated block of OIDs that it received when it opened HomeGraph in write mode. Since the graph server has already been allocated a block of OIDs, it does not have to communicate with the storage server unless it wants to. However, if HomeGraph had been opened in read mode, then no OIDs would have been pre-allocated. In that case, the graph server will have to explicitly request a block of OIDs before it can create the new subgraph. By default, the subgraph is created in the same partition as HomeGraph. The graph server creates the subgraph in its local data structures, and returns the new OID to the client. The graph server logs the create_subgraph operation to its write log. However, the write log is not necessarily flushed to disk. The graph server buffers log entries so that it doesn’t have to write to disk on every operation. The write log is flushed periodically and also when an explicit flush or close request is received from the application.

Open_Subgraph( T2, write ); /* Opens the new subgraph for writing */

The graph server retrieves the subgraph from its local cache. In addition to performing read and write operations on the cached copy of the subgraph, the graph server will also log write operations to disk.

Because the graph was opened for writing, the storage server would normally allocate a block of OIDs to the graph server for the sole purpose of creating new objects in T2. However, this will not occur if the graph server has not yet notified the storage server of the existence of T2. Two alternatives exist. Either the graph server must create T2 in the storage server and open it properly, or the graph server can borrow some of the OIDs that were allocated with HomeGraph. It can do this because T2 and HomeGraph are in the same partition. However, this is a little bit sneaky. If performance is not a problem, then the first alternative is preferable.

AnotherNode := Create_Node( HomeGraph ); /* Creates a new node in the subgraph */

Same as other create node operation. See previous example.

HG2 := Create_Hypergraph_As_Content( AnotherNode, HomeGraph ); /* Creates a new hypergraph */

The reader can assume that HG2 is created in a manner which is similar to the way that T2 was created. Notice that HomeGraph is passed as a parameter and that as a result the new hypergraph is created in the same partition as HomeGraph.

Set_Attribute( T2, 'HGRAPH', HG2 ); /* Sets T2's common attribute 'HGRAPH' to have the value HG2 */

The graph server changes the value of the attribute in its local data structures and returns control to the client. The graph server logs the Set_Attribute operation. However, the write log is not necessarily flushed to disk. The graph server buffers log entries so that it doesn’t have to write to disk on every operation. The write log is flushed periodically and also when an explicit flush or close request is received from the application.

N4 := Create_Root( T2 ); /* Creates a new root node in the tree T2 */

The graph server the new node using an OID from a pre-allocated block of OIDs. By default, the node is created in the same partition as T2. The graph server creates the node in its local data structures and returns the new OID to the client. The graph server logs the Set_Attribute operation. However, the write log is
not necessarily flushed to disk. Since the graph server has already been allocated a block of OIDs, it does
not have to communicate with the storage server unless it wants to.

\[ N5 := \text{Create\_First\_Child}( N4, T2 ); /* Creates a new node as the first child of node N4 in the tree T2 */ \]

The same as Create\_Root() except that the graph server maps the Create\_First\_Child() operation into two
lower level operations: \( N5 := \text{Create\_Node}( T2 ) \) and \( \text{Create\_Link}( N4, N5, T2 ) \).

\[ \text{Open\_Hypergraph}( HG1, \text{write} ); /* Opens the hypergraph for writing */ \]

Nothing special here.

\[ \text{Open\_Hypergraph}( HG2, \text{write} ); /* Opens the hypergraph for writing */ \]

The main question here is whether HG2 is retrieved from the cache or whether it is retrieved from the
storage server. The answer depends on whether the storage server has been notified about the existence of
HG2.

\[ HL2 := \text{Create\_Hyperlink}( N5, N2, HG1 ); /* Create a hyperlink from node N5 to N2 in HG1 */ \]

Again, nothing special. The important thing here is the purpose of this operation. HG1 is associated with
subgraph T1 and is used for storing all of the hyperlinks which come into or go out of T1. Similarly, HG2 is
associated with subgraph T2 and is used for storing the hyperlinks which come into or go out of T2. To
maintain the semantics of these two hypergraphs, the application must add the new hyperlink to both
hypergraphs since N5 is in T2 and N2 is in T1. Thus, the hyperlink is created in HG1 and the next
operation (below) adds the link to HG2.

\[ \text{Add\_Link}( HL2, HG2 ); /* Adds the new hyperlink to hypergraph HG2 */ \]

Nothing special.

\[ \text{Close\_Hypergraph}( HG1, \text{REPLACE} ); /* Close hypergraph HG1. Flush all buffers and make all changes
permanent */ \]

The hypergraph is closed. All operation buffers are flushed and the buffered operations are sent to the storage
server. First, the storage server must log all of the operations that it is receiving from the graph server and
then log HYPERGRAPH HG1 CLOSED. At this point, the storage server has stored enough information
in the log to recover from failure. However, it might be better for the storage server to go ahead and
perform the operations on its copy of the HG1. Since the user has decided to replace the original
hypergraph, the storage server can discard the original version. All future opens on this hypergraph will
receive a copy of the new version.

\[ \text{Close\_Hypergraph}( HG2, \text{REPLACE} ); /* Close hypergraph HG2. Flush all buffers and make all changes
permanent. The REPLACE parameter indicates that the changed copy should replace the original */ \]

The hypergraph is closed. The server encodes the hypergraph's data structure into a block of bytes and
sends the block to the storage server as a part of a close request. The storage server receives the block,
writes it to disk, and updates its access control tables to indicate that the object has been closed. Just before
the close operation completes, the graph server logs a close record to its write log and flushes it. Since the
user has decided to replace the original hypergraph, the storage server can discard the original version. All
future opens on this hypergraph will receive a copy of the new version. If the graph server has not notified the storage server of the creation of the new link, then it should probably do so at this time.

2.3.3. Edit the Content of the Root Node of Tree T2

Open_Node( N4, ACCESS = write, HINT = whole ); /* Open Node N4 with write ACCESS to its common attributes. No access is given to the content except for general queries such as asking whether the content is a file or a subgraph. */

Very similar to other open operations that we have seen except that a hint to the graph server has been included as a parameter. The hint indicates that the application is interested in both the attributes and content of the node (i.e., the whole node). Based on this hint, the graph server will probably retrieve the whole node before this operation completes. However, despite the fact that the whole node is retrieved by the graph server, the Open_Node() operation does not give the application a handle on the content of the node. The node's content must be opened separately via another operation.

Open_File_Content( N4, TableType, ContentHandle, Tables ); /* Gives the application a handle (e.g., filename) on the content and the associated anchor tables. The application specifies the type of anchor table that it is expecting (using TableType). If the anchor table has a different type than was expected, then the open_content operation fails. */

Since the node was opened with the "whole" hint, the graph server already has the content and does not have to interact with the storage server. When the Open_File_Content() operation ends, the file must exist within the file system of the application and ContentHandle must contain the pathname to the file. The application is expected to open and access the file using normal UNIX system calls.

Close_File_Content( N4, Tables ); /* Closes the file content of N4 and replaces the original file and anchor table with the modified versions. */

Before executing this operation, the application should finish writing to the file and close it using the appropriate system call. If the file has not been properly closed and flushed to disk, then changes may be lost. The graph server should take steps to insure that the file will not be modified further by the application.

Close_Node( N4, REPLACE ); /* Closes the node and permanently saves the changes to its content in the storage server. */

This operation permanently stores the node and its content in the graph server.

Close_Subgraph( T1 ); /* Close subgraph T1 */

Nothing special.

Close_Subgraph( T2 ); /* Close subgraph T2. Flush all buffers and make all changes permanent. */

Nothing special.