Database Management for Distributed, Multimedia, Collaborative-Writing

Almon Wesley Wear, Jr.

Technical Report 94-18

A Design Project Report
Submitted to the Graduate Faculty
of Auburn University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Computer Science and Engineering

Project Advisor - Dr. Yu Gong
Co-Advisor - Dr. Kai H. Chang
Committee Member - Dr. James H. Cross II

Auburn University
Auburn, AL 36849
October 31, 1994

Abstract

Traditional computer applications have been designed to be run by one user at a time who does work in a single medium, such as ASCII text, and very little regard has been given to the fact that people often work together. Single-user computer applications also create barriers to group collaboration and they are normally used only passively by the group to store, retrieve, and present data. If group members need to work on the same item, for example a document, in most cases they must work in an interleaved fashion to prevent inconsistency. With the recent development of computer networks and the widespread deployment of networked workstations, automating the group writing process for geographically distributed users has become feasible. There is thus a growing interest in a new generation of multi-user computer applications which assist groups of people working together, supporting collaboration even if the team members are geographically distributed. These new tools have been categorized under the name of Computer-Supported Cooperative Work (CSCW). Another new field that has emerged over the past few years is that of multimedia. Multimedia applications allow the user to manipulate information in such diverse media as traditional ASCII text, graphical text, graphics, still images, and audio and video communication. A software package which supports distributed, real-time, multimedia CSCW, known as the Distributed Collaborative Writing Aid (DCWA), is introduced. The DCWA has five major parts but the emphasis of this report is on the DCWA's distributed database, which is crucial to making the entire system work together both logically and consistently. The database's design and implementation is described from the initial prototype through the second version. The second version's database is a specialized, distributed program based on the client-server model implemented in `C++` running under 4.3BSD Unix. It provides many management capabilites unique for a multimedia CSCW environment, in addition to many conventional database functions.

Key Words : Distributed Database, Computer Supported Cooperative Work (CSCW), collaborative writing, multiuser environment, multimedia

Contents

1 Introduction

Computers are now familiar tools at work and at home and are increasingly influencing the way in which people interact due to recent advances in communication networks. Electronic mail, bulletin boards, newsgroups, distributed file systems, group schedulers, video conferencing, file transfer protocol (FTP), and "talk" services are but a few examples of computer-supported interactions. However, using computers to enhance people's communication capability has made some of the current limitations of the technology very noticeable. In particular, traditional computer applications have been designed to be run by one user at a time who does work in a single medium, such as ASCII text. Very little regard has been given to the fact that people often work together and even the scientific study of human-computer interaction (HCI) has emphasized the exploration of issues when only a single user interacts with a computer [1].

Since modern technology is complex, it is unusual for an individual to tackle the design of a major project single-handedly and usually a team is assigned to work on the project. Today it is not uncommon for documents to include contributions from many people, though usually one person is responsible for collecting the various parts and merging them together to form the (hopefully coherent) final set of documents [2]. In addition, single-user computer applications create barriers to group collaboration and they are normally used only passively by the group to store, retrieve, and present data. If group members need to work on the same item, for example a document, in most cases they must work in an interleaved fashion to prevent inconsistency. This method, however, means that all the other members are prevented from working on the document until the person is finished, even if their contribution is in a different section of the document. This method is clearly inefficient and as the need for improving productivity continues to grow, new technologies must be developed to provide active support for a team and overcome any geographical distribution of the team members [3].

There is thus a growing interest in a new generation of multi-user computer applications which assist groups of people working together, supporting collaboration even if the team members are geographically distributed. These new tools have been grouped together under the name of *Computer-Supported Cooperative Work* (CSCW) [4, 5, 6, 7]. As the term indicates, "Computer-Supported" means the use of technologies such as computers and networks and "Cooperative Work" means that the tool must provide an environment suitable for a group of people to accomplish a common goal [4]. The collection of hardware and software which supports CSCW is also known as *groupware* [8]. In the past, centralized groupware on both mainframes and PCs was explored as a means for achieving *office automation* [9] but now the focus has been shifted to the development of distributed, networked groupware for CSCW [9]. "Groupware" is also meant to be a more technically-oriented label to differentiate it from "group-oriented" products, which are simply add-ons to single-user products so that groups of people may work together using these more traditional applications [1].

In contrast, CSCW was defined by Saul Greenberg [1] as "the scientific discipline that motivates and validates groupware design. It is the study and theory of how people work together, and how the computer and related technologies affect group behaviour." However, CSCW is not a particularly well-defined area due to its youth and multi-disciplinary nature and, for the moment, the term CSCW simply serves as a useful forum for researchers with different backgrounds to discuss their work. For example, CSCW researchers come from a variety of specialized fields, such as computer science, psychology, cognitive science, anthropology, sociology, management and management of information systems. These fields contribute different perspectives on group interaction, methods of acquiring knowledge about groups, and suggestions on how the group's work could be better supported.

Even the term "Computer-Supported Cooperative Work" has been considered inappropriate for a variety of reasons [1]. For example:

- *computer* – technologies other than computers, such as video, are considered part of the CSCW domain.
- *supported* – support may be offered to the group as a whole but this may actually be disruptive to the activities of particular individuals.
- *cooperative* – the social process can include not only cooperation but also aggression and competition.
- *work* – casual and social interaction should be supported too, since they are considered by some to be a vital precursor to the work process itself.

Another limitation to users has been the reliance on applications which support only one type of information medium. Another new field that has emerged over the past few years is that of *multimedia*. Multimedia applications allow the user to manipulate information in such diverse media as traditional ASCII text, graphical text, graphics, still images, and audio and video communication. The main reason for the growing interest in multimedia systems has been the rapid development of networking technology and with the completion of a Broadband Integrated Services Digital Network (B-ISDN) based on fiber optics, larger amounts of all kinds of data (such as text, audio and video) may be transmitted through a single channel efficiently and cheaply [10] as compared to the networks of just a few years ago.

This report introduces a tool known as the *Distributed Collaborative Writing Aid* (DCWA) which was developed to provide multimedia CSCW on workstations connected to the Internet and running the UNIX operating system. The DCWA can help users cooperate on any writing task (such as programming, report writing, note taking, etc.) logically, conveniently, and efficiently. The coding of the DCWA has been divided into five major parts and this report's emphasis will be on the DCWA's distributed database, which is crucial to making the entire system work together both logically and consistently. The next section explores the features of some related systems and then compares and contrasts the DCWA with them.

2 Literature Review

"Distributed systems have traditionally been viewed in terms of support for cooperation between a number of computers connected by a network." [11] More recently, they have considered the interactions of the users but have still taken a "systems-oriented" approach to control by making any distribution problems transparent to the applications. For example, in most distributed systems the problem of shared access to resources, is usually dealt with by masking out the existence of other users. Thus, sharing is a transparent activity where each user is unaware of the others but this clearly contradicts the principle and needs of CSCW [11]. Groupware for geographically distributed users is usually based upon an architecture that allows multiple processes to communicate with each other [12].

Therefore, a fundamental requirement for CSCW is the provision for message exchange among the participants and it has been suggested that messages should be organized in a structured way to achieve efficiency [13]. That is, protocols must be designed to discriminate between cooperation information and messages of little relevance, e.g., the cognitive filtering in Information Lens [14] and the semi-automatic agents in Object Lens [15].

A second fundamental requirement is the provision of mechanisms for information sharing and cooperation [16]. Most CSCW environments stress the need for the distinction of public and private information. It has been suggested that sets of nodes for both shared and individual work should be supported [2]. On the otherhand, it has been argued in [17] that information to be used in cooperative work needs to be taken out of the limit of a "personal wall". It was suggested that all users should have common access to all information.

2.1 Background

Cooperation within groups of human beings has taken place since prehistoric times, so the concept of "group work" is nothing new. The scientific study of group work can be traced back to the beginnings of sociology and psychology at the turn of the century and a number of concepts have become established in the research that has been done in these fields [18]. For example, the group's "task" (i.e., its reason for existence) is usually divided into two parts :

- Its goal – that is, what is to be achieved by the group.
- Its process – that is, the method by which the group goes about achieving its goal.

Furthermore, the group process may be divided into those behaviors (called *task behaviors*) which are aimed at achieving the group's task and those behaviors (called *maintenance behaviors*) which are aimed at maintaining the group as a cohesive unit. These two types of behavior are antagonistic and the group process usually involves both of them as progress is made towards the goal. Someone who understands group processes and can therefore assist a group to understand its problems, and find solutions for them, is a valuable asset to any group. Despite the importance of the role of such a facilitator in improving the effectiveness of group work, it has been largely neglected in CSCW research. Communication over a computer network will affect the interaction between the group members in unforeseen ways until it is well understood. Likewise, the needs of a facilitator as he/she performs his/her duties are not yet well understood either. Hence, the exclusion from most CSCW systems. [18]

Teamwork introduces the social problems of organization, coordination, cooperation, and communication and these problems are accentuated when the members of the group are geographically distributed. In [1] it is discussed that there have been suggestions that the current approaches to groupware are mechanistic, i.e. "based upon a task-oriented approach that can be modeled by a machine." Others have suggested, however, that groupware should be based on "the social theory that human systems are self-organizing and arise out of the unrestricted interaction of autonomous individuals." That is, groupware should not directly regulate the actual meeting process but instead should allow the group itself to define its own conventions, structures and constraints, possibly changing them over time.

Some researchers argue that groupware should impose social models of interaction on the group, as well as provide methods for keeping the group directed towards its goal, enforcing roles, and making the group more efficient. Others argue that social protocols should be determined by the group members themselves and still others argue that it should fall somewhere in between [12].

Multimedia

A central concept of interactive, multimedia documents is the notion of supporting "nonsequential viewing", as well as "nonsequential authoring." A result of these nonsequential operations is that there is not a predefined order in which the different segments of the document should be handled. One way of perceiving a nonsequential document is as a collection of nodes and links, "where the nodes represent the document segments and the links capture the access routes to the nodes" [19]. That is, the heterogeneous media are easily accessible since they have been interlinked via an homogeneous linking mechanism [2].

Clearly, the construction of such an access graph is an important part of the document authoring/editing process. However, it is also an important part of the document reading process as well, which involves navigating through that access graph. In many cases, the navigation aids made available to the group are crucial since many documents are exceedingly complex and the users may have difficulty finding their way through them without any navigation tools [19].

Database Systems

CSCW systems have benefitted from the current technological advances in computer networks and database management. Networks and advanced communication facilities provide for communication between distributed sites and database management systems allow data to be accessed/shared by multiple users [20]. The main challenges associated with designing a database for CSCW are :

- Implementing the partitioning method which will break the documents into manageable sections.
- Maintaining this partitioned structure as changes are made by each user.
- Updating the master copy as needed.
- Determining if two users are trying to modify the same section.

In traditional database systems, the ability to retrieve materials depends on the system and is usually based on some sort of key. However, these keys are static and may be extremely standardized or so personal that they are useless to all members of the group except for their creator [2].

Also, concurrency control (needed to mediate access to any sharable objects) is usually achieved through simple locking, though other systems use transaction mechanisms [12]. Floor control or a turn-taking mechanism is another way to provide access control to the shared objects and it has been recommended that systems support many such floor control policies so as to suit the various users' needs [12].

Until recently, most multimedia objects have been stored in files by using "document management systems" to store the documents' index and attribute information, where one of the attributes indicates the physical location of the file. However, these systems usually keep journal files to provide for recovery after system failures. Unfortunately, this presents a disadvantage to using commercial, relational database systems since the database journal files tend to be bottlenecks since every transaction is recorded in the journal [21]. It is also important to realize that "documents rather than simply numerical data will be the overriding user need (and database load) in the 1990s." [22]

2.2 Classification of CSCW Systems

In the field of collaborative writing, the collaboration among coauthors may be classified into two categories : *shared mind* and *division of labor* [7]. In the first category, multiple authors edit the same file at their own locations. In the latter category, a task is divided into mutually independent subtasks for coauthors, and a coordinator integrates the contributions. These classifications are similar to asynchronous and synchronous collaboration. Asynchronous systems allow only one author to actually work on the document at any given point in time. Synchronous systems, on the otherhand, allow two or more authors to work on a document at the same time [20].

Another characteristic which may be used to classify CSCW systems is the form of cooperation which is supported [11] :

- Some systems require the presence of all the cooperating group members.
- Some systems are designed to allow cooperation without the simultaneous presence of all group members.

- Some systems contain both of the above, such that all members are needed for some operations but not for others.

A final method of classification is how widely the users may be distributed geographically [11] :

- *Co-Located Systems* require the presence of all users in one place, such as a meeting room.
- *Virtually Co-Located Systems* do not require that all the participants be in the same room. For example, they may be connected via a LAN.
- *Locally Remote Systems* provide high-bandwidth real-time accessibility between participants, such as in a high-speed WAN.
- *Remote Systems* assume only minimal accessibility between participants, such as email.

Examples of some CSCW systems classified under these last two categories are seen in Figures 1 and 2 from [11].

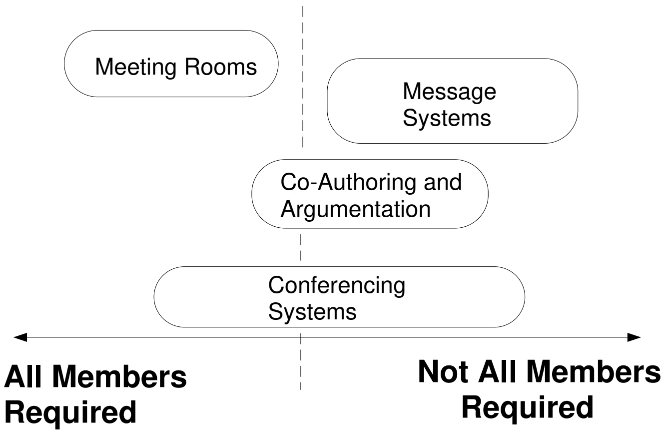


Figure 1: Forms of cooperation in CSCW systems

2.3 Related CSCW Systems

Interest in the area of CSCW has grown rapidly since the first CSCW workshop in 1984 yet there are still relatively few successful products [2]. Of course, if a CSCW tool requires the users to put more of an effort into learning to use it than a single-user system would, then

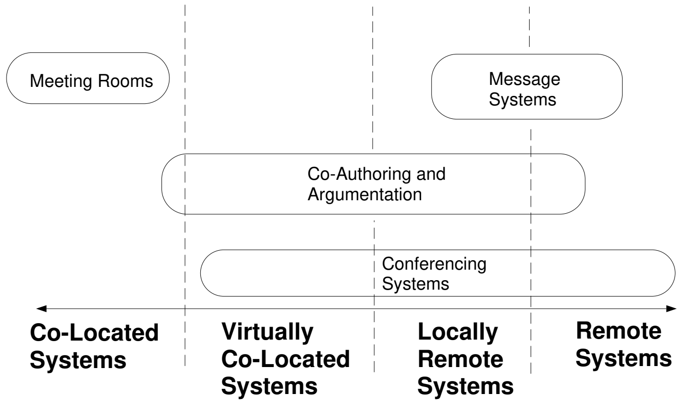


Figure 2: Geographical nature in CSCW systems

the users will reject the system [20]. It has also been pointed out that in supplying aids for collaborative work, a system should not hinder the productivity of an individual's work with respect to traditional applications. In other words, adding collaborative support should not affect an author's ability to write by him/her self. [20]

An experimental system providing asynchronous collaborative writing for a small group has been reported in [5]. The system uses email as the communication media and the text is represented in the ASCII format. The advantage of this approach is that the impact of the geographical locations and the computer variation among the team members is kept to a minimum. However, since only email is used, referencing comments or making changes to existing files may be difficult. An interesting observation made by the developer was that the success of collaborative writing depends heavily on the mutual confidence and trust among the team members.

The NoteCards system of Xerox PARC allows multiple users to open and read the same node but only one of the users may modify the node's content at any given time. It is said that this limitation leads to drafts of the document being passed back and forth between the authors. [20]

CSCW can also be accomplished through multimedia communication channels. The MERMAID [23] and SPIN [24] systems are prototypes that provide real-time conferencing environments for geographically distributed participants by using synchronous textual, audio, and video communications. One of the drawbacks associated with these systems is that the cost is far beyond what most users can afford. A simplified version of a conferencing service is to use extended textual bulletin boards [16]. In addition to providing a conferencing service, the COGNOTER system [25] also collects and organizes ideas from its participants for discussion. It is one of the piloting systems that provides the WYSIWIS (What You See Is What I See) capability.

The principle of hypertext technology is to manage a structure of text, graphics, and other media in a network of nodes. A hypertext system provides comprehensive services for document writing and reviewing and is normally distinguished by two features — links and windows [26]. A document is organized into nonlinearly linked nodes. A link may indicate a referential or an organizational relationship. Windows are used to provide the display and editing services of the content of a node and also allow a user to traverse the links. The feature of linked node organization is not limited to the display, however, since there must be a one-to-one node correspondence in the database too. Although hypertext provides new possibilities for authoring and design, it suffers from the disadvantages of user disorientation and cognitive overhead. The technology is mainly for a single user working alone at any given time. There are, however, a few hypertext systems that are designed for cooperative work.

Quilt [27] is a hypertext system that provides coauthoring services. The users are identified as either coauthors or commenters. It allows all users to comment on a document but only certain privileged users, i.e., the coauthors, can modify the document. The For-Comment system [28] provides a group editing environment and allows up to 16 reviewers to comment on a document. However, only the original author can actually modify the document.

The Aquanet system developed at Xerox PARC is a hypertext tool which facilitates collaborative knowledge structuring. It provides a "What You See Is What I Did" (WYSIWID) view on the shared structure but does not provide synchronous shared views or communication facilities [20].

The rIBIS [29] and SEPIA [20] are synchronous hypertext systems providing various collaboration modes. In the independent mode, users may work on their own tasks without interfering with each other. In the loosely-coupled mode, users may share certain public information while working on their own tasks and in the tightly-coupled mode, users share the same view and resources, e.g., mouse and file, which are strictly controlled to avoid conflicts. Major improvements of SEPIA over rIBIS include automatic mode switching and the use of composite nodes. SEPIA also provides an audio communication channel between participants.

Office Document Architecture (ODA) [30, 31, 32] is the International Standards Organisation's (ISO) international standard ISO 8613 and its purpose is to facilitate the electronic interchange of documents between different document applications, such as Word for Windows, WordPerfect, and IslandWrite. Though not specifically for CSCW systems, it specifies that a document should be separated into the following three parts :

1. The logical structure with elements such as chapters, paragraphs, footnotes, figures, or any other object a system may use to logically partition a document's content.
2. The layout structure with elements such as pages, columns, etc.
3. The actual contents of the document.

Clearly, this is related to the graph structure as used by some CSCW systems. Also, many single-user systems separate the properties of an object from its view on the screen. It has been argued that such a separation is critical in groupware and a consequence of this method is that users can have multiple perspectives on the same data [12].

Finally, some systems supporting CSCW collect information from the participants and then suggest appropriate courses of actions. The SYBIL system [33] collects and coordinates options about a specific design topic from members of a hardware platform project team, and then suggests goals and subgoals to them. The Office Works system [28] arranges meetings for a group of participants by checking each participant's schedule.

2.4 The DCWA

The DCWA described in this report was designed to provide specific solutions to collaborative writing problems by addressing many of the issues identified in the literature. It has been designed to be groupware for geographically distributed users and is based upon an architecture of multiple communicating processes, as explained in more detail later. It has both asynchronous and synchronous aspects but does not require the simultaneous presence of all group members. The tool is asynchronous in the sense that all users may freely join, quit, and rejoin the group at any time during the writing process. It is synchronous in two respects. First, the group members may modify the same file at the same time and secondly, any team member may view any other team member's working area. The changes which are seen being made to the other member's working area are synchronous with the actual modifications occurring at the remote site, thus supporting the WYSIWID style.

Furthermore, the DCWA was not designed to research social problems/models and, like other CSCW systems, does not support any distinction between the various users, such as a facilitator or leader. However, these problems could certainly be studied using the DCWA at some later stage of development. Also, the DCWA does not directly regulate the actual collaboration process but, of course, some groups may feel that it imposes itself on them since they may have needs not yet addressed by the DCWA working group. For example, it does not collect information from the participants and then suggest an appropriate course of action.

One goal of the overall project is to allow the users to be *Locally Remote*, though to date testing has only been done in *Virtually Co-Located* environments. In general, it supports the concept of "no personal walls" but there are means to limit the scope of each user if the group desires to enforce such a style of collaboration. Finally, it does degrade gracefully to a single-user system if only one group member is currently working.

The composite node concept of [20] has been adopted to organize the file structure. A tree-like hierarchical structure is used to define the non-overlapping working spaces for the users and guarantees a conflict-free, collaborative environment. This structure normally reflects the organizational structure of the coauthored documents, such as the separation of the documents into chapters, sections, subsections, etc., and does not carry much information about the contents. It is also similar to the ODA's division of a document in that the group is allowed to define their own logical structure (the ODA's first part) and the content (the ODA's third part) is represented by the leaf nodes. As the DCWA evolves beyond ASCII text, the inclusion of the layout structure (the ODA's second part) will become important. For example, using "bounding boxes" to insert a graphic figure in text. This layout structure may also become important if and when the DCWA working group decides to implement hypertext links.

Not only does the DCWA allow the group to define their own structure, rather than having to work with an application-defined structure, it also allows them to specify a semantic network which is "overlaid" upon the organizational structure to allow more meaningful migration from one node to another. Clearly, the organizational structure follows the "division of labor" principle, and the provision of the semantic network follows the "shared mind" idea. These two ideas are combined in the DCWA by allowing the nodes of the organizational structure to take on certain attributes.

In addition to the text editor of most conventional collaborative writing systems, the DCWA also provides a graphical editor. The distributed, multimedia database organizes both the textual and graphical information according to their structural relationships, as well as their semantical relationships. Users may navigate through the files logically by using a search facility, which queries the node attributes and limits the user's view to only those nodes which meet the criteria of the query. This is a useful facility when either the organizational structure becomes large or the group member is new and thus unfamiliar with the structure that the group has established. Finally, all of the attributes (keys) except for one may be used in the search criteria, thus hopefully preventing "useless" keys. Though most of the attributes are static, the values of some of them may be modified by the group whenever they choose and the DCWA also supports the concept of dynamic keys via the use of user-defined attribute/value pairs.

Communication among users is accomplished through a real-time, scrollable, textual "talk" window, which is different from the audio channels provided by most systems [20, 24]. A textual message is less intrusive and since it remains on the screen, it makes referencing the discussed subject easier, as well as allows parts of the dialog to be copied and pasted elsewhere, for example into the contents of a text node.

Finally, the user interface not only displays the user's working area but can also display any other group member's working area. It uses both the WYSIWIS (What You See Is What I See) and WYSINWIS (What You See Is Not What I See) styles. The WYSIWIS style is typical for synchronous cooperation where members must see changes instantly, while the WYSINWIS style is important in case information must be tailored before being presented to a remote user, such as seeing a 3D graphics object from different perspectives. A mixture of these two styles appears to be most useful.

As stated in [7], there are currently very few systems supporting collaborative writing. Therefore, it is still premature to provide solid guidelines for such a system's development. However, some requirements for collaborative writing have been suggested [6, 16] :

- Provision of communication and exchange of information among authors.
- Mechanisms for sharing information to allow cooperation.
- Management of inputs and outputs for users.
- Mechanism for role control of the team members.

In short, they should facilitate what the group can do at a face-to-face conference in a much cheaper way. The DCWA includes most of these characteristic requirements, as well as many additional and novel features. Many problems remain open in a distributed environment but the most noticeable ones are *group organization*, *multicasting*, *distributed* *database management*, and *user interface layout*. These problems are mutually dependent and cannot be dealt with in isolation. Detailed requirement analysis for effective CSCW has been reported in [4, 5, 6, 7].

Almost all existing CSCW tools incorporate some database capabilities and a large number of them even directly utilize commercial database systems. In the design of the DCWA, even though an Oracle 6 system was available, it was not used, mainly because it conflicts with the goal of allowing users in different file systems to work together. In addition, unlike other software systems, database systems have widely diverse interfaces for application programs and no commercial product satisfied all of the DCWA working group's needs. Since the database of this project would also need to enhance the capabilities of any standard database, the decision was made to build a specialized, distributed database for the DCWA. The specific solutions to the problems associated with database maintenance for CSCW, as designed and implemented by the author, will be presented in later sections but an overview is given here :

- The partitioning method used is the hierarchical tree structure described above.
- The users may modify this structure and the changes are reflected at each site.
- The users may make changes to a node and save them locally. Finally, they commit the changes to update the master copy.
- A locking mechanism was implemented to prevent two users from modifying the same node.

The next section describes the preliminary work done on the DCWA in general and its database in particular.

3 Preliminary Work : The Prototype

3.1 Statement of the Problem

The overall purpose of this project is to develop a software package which supports distributed, real-time, multimedia CSCW. That is, it will allow a group of people working at remote sites, connected via the Internet, to collectively work on multimedia documents at the same time, with changes made by one of the users reflected on the other users' workstations in real-time.

This is, however, the long-term goal of the system and it will actually be implemented in several stages, thus evolving from an initial prototype to the final system. This plan makes the problem more manageable and allows evaluations after each phase's implementation. The method is thus similar to the *spiral model* [34]. First, a prototype was rapidly implemented and then evaluated. A second version was then designed and implemented which not only eliminated any problems found with the prototype but also incorporated many new features as well. The scope of the author's work was up through the implementation of the second version but the work continues. (That is, the second version will be evaluated as well and a third version will be designed, etc.) See Appendix B for a timetable showing some of the highlights of the work to date.

The main requirement of any CSCW tool is to allow a group of people to work on the same object at the same time. Therefore, the first task was to decide on the format of the sharable object. After discussing the pros and cons, the DCWA working group decided to completely remove the multimedia component of the system and chose ASCII text files as the sharable object for the prototype.

The DCWA working group also decided to allow the users to coauthor only one file at a time and the scope of the prototype was narrowed down even further by allowing only two users per group. The final simplification was the removal of the distributed component of the system in the sense that the workstations would be assumed to be connected by a Local Area Network (LAN) rather than a Wide Area Network (WAN). Thus the shareable file is assumed to be accessible via a local Network File Server (NFS). These final simplifications made the real-time component trivial but messages were still exchanged between the two workstations to allow each user to see the other's work, since even though they were connected via a LAN they may not necessarily be in the same room together.

3.2 Logical View

One of the major requirements of the database is to prevent conflicts when the users make modifications to the file. Normally, a user views an entire file as one logical object but now that object must be shared between more than one user at a time. The solution was to divide the text file into separate logical units, called nodes, even though the file is still stored on disk as one physical unit. These nodes are arranged in a general, acyclic, unweighted graph, called the *logical view*. The logical view allows the system to permit users to select and edit different parts of the file at the same time. Conflicts are prevented because the database "locks" a node when a user selects it and prevents the other user from selecting that node until the first user releases it. Although UNIX file and record locking exists [35], they were not used since those capabilities have been turned off by the System Administrators at the College of Engineering at Auburn University.

An important requirement of the DCWA is to allow the users to define the logical structure themselves rather than following a structure defined for them by the tool. Therefore, they may define, modify and save their own organization (e.g., sections, paragraphs, functions, etc.) for the text file. In the prototype, another module (the user interface) actually provided the logical view and node manipulation capabilities instead of the database. However, these capabilities are discussed here since in the second version of DCWA the database was responsible for all these capabilities, except for the actual presentation of the information to the user. A discussion now will also facilitate the discussion in later sections concerning the enhancements that were included in the second version.

In the prototype, when a file is opened for the first time under DCWA, the group must define the logical view associated with the file. Although this process will require some initial setup time before group editing can begin, it allows the group to define their own logical view rather than having to work under some application-defined structure.

First, the group needs to divide the root node (which initially represents the entire file) into different child nodes, each containing the contents of a specific part of the file. The group must also assign a unique label to each node at this time. This process is open-ended and can be repeated for as many levels as desired.

DCWA also provides the ability to merge sibling nodes back up into their parent node. This process is needed because the division of the nodes may have been done incorrectly or may need to be updated. However, this capability allows the group to merge only nodes without any child nodes. Thus, it is similar to the UNIX `rmdir` command, where before removing any directory, it and all its subdirectories should be empty. Thus, if several levels need to be merged back into one node, several uses of the merge facility will be needed.

The following restrictions were imposed on the group while defining or modifying a logical view in the prototype :

- A node label of "@" is reserved and may not be used.
- Node labels must be unique. That is, the same label may not be used for different nodes within the same logical view.
- The contents of the file are represented by the leaf nodes. Non-leaf nodes contain no information and are only used for the purpose of connecting the tree together logically.

- The leaf nodes may not represent overlapping parts of the file but rather they should represent juxtaposed parts of the file. This constraint is required to guarantee the non-conflicting work spaces for all users.

- The entire file should be represented by some node. That is, there should not be any part of the file which is not represented in the logical view.

- The "granularity" of the various nodes may differ and are defined by the group. In fact, nodes may represent any number of characters from 1 to 32K. (The maximum size restriction is due to the `size_t` parameter of the standard C function `malloc()` but a positive side-effect of this is that it limits the sizes of the nodes to something which is reasonably efficient for transfer across a network.)

When the logical view is the way the group wants it, it should be saved. If it has not been defined exactly the way the group wants it for some reason or should the group need to change it at some later time, DCWA also provides the capability to modify the logical view. It is saved in a special file called `filename.struc` in the same directory. For example, if the group file is called `lab1.c`, the structure is stored in `lab1.c.struc`. It is important to protect this file, since any accidental changes to it will cause the corruption of the node structure and thus cause DCWA to manipulate the nodes erroneously.

Once the logical view is saved, it becomes associated with the file and will be loaded automatically when the group wants to edit that file from now on. Also once it is defined, group work may begin in earnest. Either user may select any leaf node and edit the text associated with that specific node. However, the two users cannot select the same node at the same time. Once a user has selected a particular node, he may monopolize it for as long as he wishes (as far as the tool is concerned.) Other users are prevented from selecting that node, though they can view its contents even as it is being edited. DCWA has been designed for groups of "friendly" collaborators so this monopolization should not be a concern. See the findings of [5] where it is discussed that the success of collaborative writing depends on the mutual confidence and trust among the team members. Finally, when a user wants to edit a different node, the previous node must be saved or discarded. If discarded, the node's content is not changed by the database but remains as it was before the user selected the node. If the user chooses to save his/her edits, the database writes the new contents of the node to the file without affecting any of the other nodes.

A user's selection of a node is normally displayed on both users' screens by changing the color of the node on the display of the logical view. However, due to network delays it is possible that the two users may decide to edit the same node, thinking that the other user is not editing it. To prevent this condition from producing conflicting edits of the node, the database keeps a time stamp associated with the user's selection and informs the user interface when this condition occurs. The user who selected the node last will be informed of what happened and will not be allowed to continue editing the node.

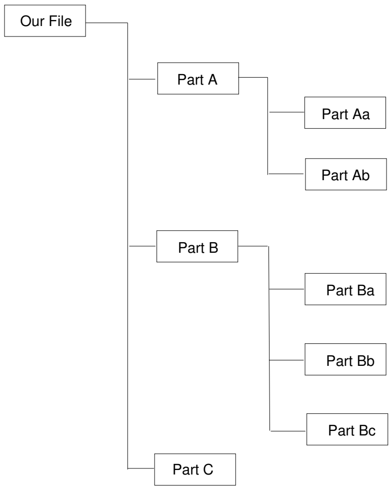


Figure 3: A group's logical structure

The logical structure is a hierarchical, tree-like structure with leaf nodes containing the actual contents of the file. An example is seen in Figure 3. The intermediate nodes serve as logical connectors and may even be used as indicators of the scopes of the users. The information pertinent to the logical structure is stored separately from the text file so that the continuous text file can be used for any other purpose. Some CSCW systems store the textual and structural information together but the choice of storing these two types of information separately was based with future expansion in mind. For example, in a software development environment, users may rely on the `makefile` to define the logical structure of the files (i.e., modules of a large program) but it is well-known that the program structure encoded in a `makefile` may possess non-tree structures (e.g., it may be a "forest" or a Directed Acyclic Graph.) Therefore, separate storage of the two types of information allows a logical structure that was compatible with an earlier version of the DCWA to be easily upgraded to a newer version. The upgrading can be done, for example, by modifying the logical structure only, without having to touch the text file(s).

In Figure 3, a logical view is shown as presented to the user by the user interface module of DCWA (see below.) This structure is similar to DOS's directory strcuture as presented in its shell tool. For those more familiar with graphs, the following two structures are equivalent :

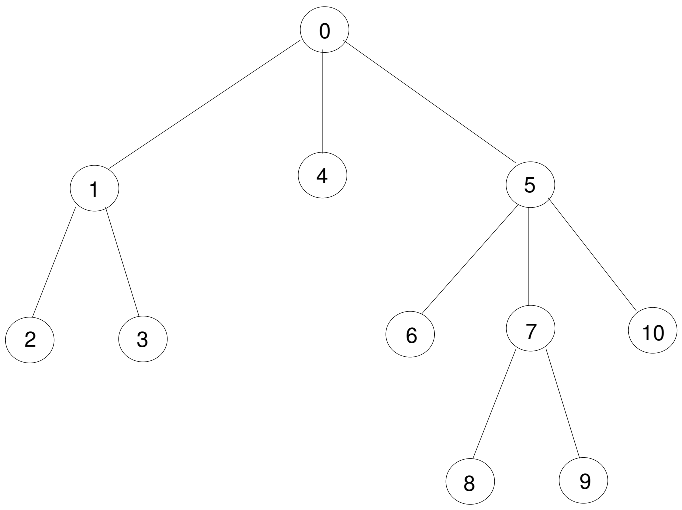


Figure 4: A Sample Graph equivalent with the Logical View below

```
0 ____
     |
     | - 1 ____
     |         | - 2
     |         |
     |         | - 3
     |
     | - 4
     |
     | - 5 ____
               | - 6
               |
               | - 7 ____
               |         | - 8
               |         |
               |         | - 9
               |
               | - 10
```

3.3 The Users' Working Areas

Since the prototype is only dealing with two users per group, there are only fives areas of the file which are of concern, as seen in Figure 5. These areas represent the following :

1. From the beginning of the file to the start of user 1's working area.
2. User 1's working area.
3. From the end of user 1's working area to the beginning of user 2's working area.
4. User 2's working area.
5. From the end of user 2's working area to the end of the file.

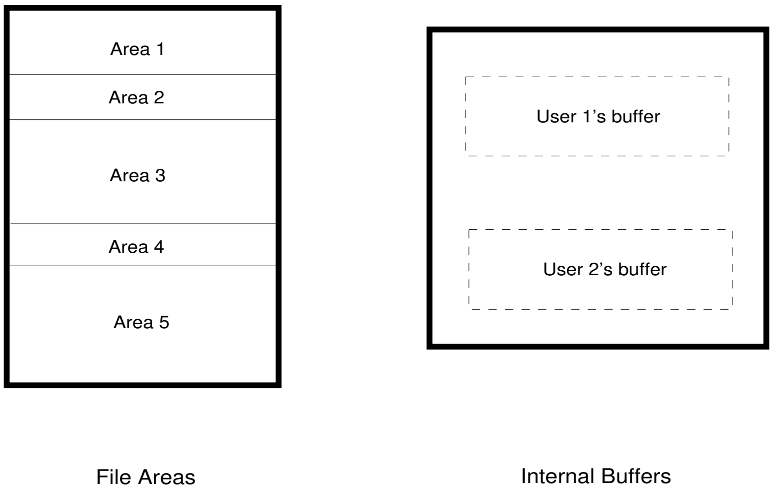


Figure 5: Working Areas and Buffers

Here "user 1" and "user 2" do not refer to the order of joining the group, or starting DCWA, or even temporal selection of a node but simply refer to the fact that in a linear file one user's working area will be *before* the other's. Areas 2 and 4 will be one node each but areas 1, 3, and 5 may be any number of nodes, including zero. (For example, area 1 would not represent any nodes if user 1 selected the node associated with the beginning of the file.)

When a user selects a node, the contents of that node are stored in a buffer by the database, also seen in Figure 5. The only concern now is to save the correct sequence of areas and buffers when a particular user wants to save his/her edits.

When user 1 wants to save :

- Areas 1 and 3 - 5 are read and stored in temporary buffers.
- Area 1 is rewritten to the file.
- User 1's buffer is written to the file.
- Areas 3 - 5 are rewritten to the file.

When user 2 wants to save :

- Areas 1 - 3 and 5 are read and stored in temporary buffers.
- Areas 1 - 3 are rewritten to the file.
- User 2's buffer is written to the file.
- Area 5 is rewritten to the file.

After each save, the database on the other user's host (see below) is informed of the new start and end points of the user's node and from these is able to recalculate the start and end points for the areas occurring after that node.

3.4 Prototype's Design Requirements

The main reason for all the above simplifications was the need to rapidly implement the prototype to prove that such a system was feasible. It was not the intention of the prototype to provide a full range of services for collaborative writing. Instead, it was to serve as an exploratory platform to examine and discover features and new requirements. However, it still provided many important CSCW features, including coordination, user-communication, logical structures, and real-time WYSIWIS (What You See Is What I See). Future versions of the system will remove these restrictions and progress towards the system's long-term goal. The following is a summary of the requirements of the prototype :

- The document which the group coauthors will be a standard ASCII text file.
- Provide the capability to allow the group to create, modify, and save a logical view of the file.
- The leaf nodes of the logical view may not represent overlapping parts of the file but rather they should represent juxtaposed parts of the file.

- Only one document may be collaborated on per editing session.
- A user may open a file, select a node to edit, edit it, and then save the node with the assurance that the system will prevent editing conflicts between himself/herself and the other user.
- A user may monopolize a node for as long as he/she wishes.
- Allow a user to select and switch between different nodes which are not being edited as he/she wishes.
- Allow a user to view the other user's editing as it happens.
- Only two users will form the group.
- Only one user per host is allowed.
- LANs and local NFSs are assumed.
- Provide the users with the ability to communicate with one another over a "talk" channel.

3.5 Overview of the DCWA Prototype

The prototype was designed, implemented, and then all of its main functions were tested using many practical and large text files (60 Kb). It was found to be fully functional and some of the details will now be given.

The DCWA prototype has been divided into five independent processes. The first is the DCWA daemon process, which is assumed to be running in the background of all the machines which may potentially be used by the group members. However, if it is not, a user may start the daemon by typing :

```
cscw_daemon
```

Since the users should not have to rely on another medium (e.g., telephone, email, or face-to-face communications) to negotiate a schedule for starting the tool together, a mechanism was needed to allow them to initiate the DCWA in a totally asynchronous manner. Any user must be able to initiate the software at any time without the knowledge of the work schedule of his/her partner. If, however, a user is the only group member currently working, then the software should gracefully degenerate to a single-user editing tool. The DCWA daemon is responsible for this asynchronous start of the software, as well as establishing any initial socket connections (see below). A user begins by typing :

```
cscw machine_name other_user's_machine_name
```

This creates a controller (CR) process, which forks off three child processes on the host, namely the Graphical User Interface (UI), the Database (DB), and the Network Access (NA) processes. Communication between these sibling processes is accomplished using UNIX pipes and since pipes are unidirectional, a pair of pipes must be established between every pair of these processes. Communication with the other user's host is accomplished via an Internet socket. Therefore, prior to forking its child processes, the CR creates all the pipes, as well as the socket. After forking its child processes, the CR suspends itself until the user wants to quit, when these connections are torn down.

When the other group member starts DCWA, similar processes are created on that host, which results in the configuration shown in Figure 6. As stated above, the prototype does not allow more than one user per host.

The task of the UI is to provide a graphical means for the user to select the file to be worked on, select the area of text which is of immediate interest, as well as providing the traditional text-editing capabilities. It can also display the other user's selected area, as well as periodically update it to show the other user's edits of the node. For more information about the UI process see [36].

The task of the DB is to maintain the consistency of the users' modifications to the file and determine if the two user's try to select the same node at the same time. (See the section above on the users' working areas.) Further details of its operation will be given later.

The task of the NA is to provide communication to the other user's host for its sibling processes. For example, if the DB process on one host needs to send a message to its peer DB process on the other host, it sends the message to its sibling NA process over a pipe instead. That NA process sends the message to its peer NA process on the other host via the socket. Finally, the remote NA process passes the message to its sibling DB process over a pipe. For more information about the NA process see [37].

There are many reasons for separating the functionality of the DCWA into these independent processes. On many occasions, the tasks associated with the processes need to be performed concurrently. For example, when a host is saving the changes made to a

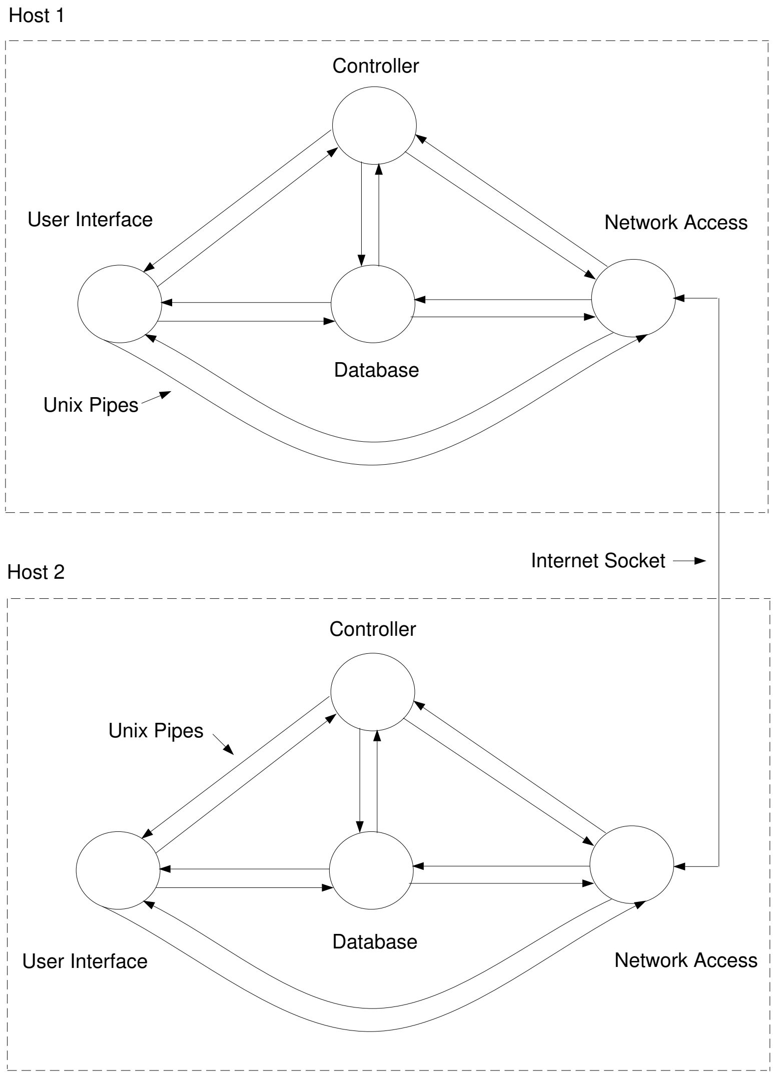


Figure 6: Configuration of the DCWA's prototype

node's content, the UI should still be able to interact with its user. This is achieved by the concurrent DB and UI processes. Similarly, when the UI process is interacting with the DB process to record local modifications to the file, messages from the remote host should still be received. This concurrency is achieved by a separate NA process.

However, if a large number of communicating processes time-sharing a single CPU were used, the responsiveness of the software would most likely be hindered by the overhead spent in communication among the processes. The prototype has only three main processes running concurrently (the UI, DB, and NA) but they cover most of the functional capabilities expected of a CSCW writing aid.

Since the progresses of the processes are asynchronous, they must exchange messages to ensure that the entire system functions logically. The three main processes all follow an event-driven principle as shown in Figure 7.

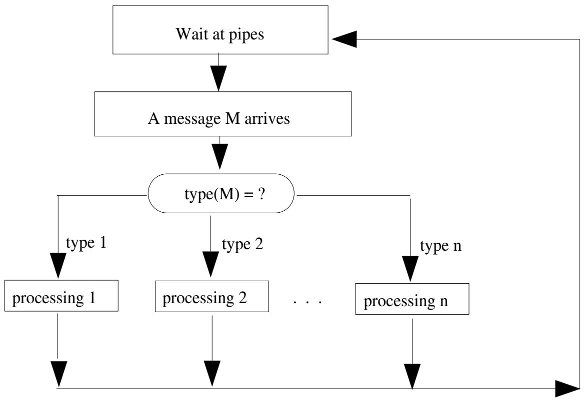


Figure 7: The Event-Driven Design of Processes

Each process suspends itself if it does not receive any messages from the other processes. Upon reception of a message, it wakes up to provide the appropriate service(s). Since all processes follow this algorithm, what differentiates a given process from the others is the message types it responds to (according to a protocol) and the corresponding function it provides for each message type. This event-driven strategy makes the DCWA an open-ended system in the sense that additional features may be provided by adding new message types, protocols, and functions into the relevant processes.

The message structure used in the prototype had three parts : a code, the sender, and the data. The value of the code is used to determine the purpose of the message. The sender field allows the receiving process to know which process was the originator of the message. (Examining which pipe the message was received on is insufficient for this purpose since, for example, the DB on one host will receive messages from both its peer DB on the other host and its sibling NA process on the same pipe.) Finally, the data field contains the character string to be communicated, which may be empty.

3.6 Overview of the Prototype's Database

The prototype database contained the following four modules (see Appendix C for the complete structure charts) :

1. `db.c` contains a function called `db_process()`, which is the function called after the DB is forked off by the CR process. It closes the socket and any unused pipe ends, determines its process id, receives its sibling processes' ids from messages sent by the CR, and finally calls `db_main()`.

2. `db_main.c` contains a function called `db_main()`, which calls an initialization function in `db_log_view.c` then goes into an infinite loop which uses the UNIX `select` system call to determine if there are any messages in any of the pipes. When a message is received, it is read and passed to `db_process_message()`.

3. `db_proc_msg.c` contains a function called `db_process_message()`, which processes the event-driven messages based upon the message code. That is, it contains a C switch statement with each message code that the DB responds to as a separate case. (The default case of the switch statement is used to ignore any messages which the DB should not respond to.) Depending upon the code, functions in `db_log_view.c` may be called, information may be sent to the DB's sibling processes, and/or information may be sent to its peer DB process on the other host via the NA process. (The complete communication protocol of the DB process is given in Appendix D.)

4. `db_log_view.c` contains functions for maintaining the logical view of the document and saving the users' changes to disk. It defines a C structure, called `document`, which contains the following fields :

   - A file pointer to the group's file.
   - A character pointer to hold the file name.
   - An integer used to indicate whether or not the two users opened the same file. (The prototype DB was designed to allow the two users to work on different files at the same time but the other processes of the DCWA prototype did not support this feature.)
   - An array of two long integers to store the starting point (in bytes) of each user's working area, relative to the beginning of the file.
   - An array of two `time_ts` to store the time when each user picked his/her working area.
   - A long integer to hold the file length.
   - An array of two character pointers used as buffers to store the contents of each user's working area, as discussed above.

- An array of five character pointers used as buffers to store each of the five areas, as discussed above. The buffers for areas 2 and 4 point to the appropriate user buffer above.

This file contains the following functions which operate on a variable, called `doc`, which is of type `document` :

- `db_init_doc()`, which initializes the document by allocating space, setting the handle, name and buffers to NULL, and then setting the other fields to N_A, which was defined as -1.
- `db_open_doc_read()`, which opens the document for reading.
- `db_open_doc_write()`, which opens the document for writing.
- `db_get_doc()`, which returns the entire contents of the document.
- `db_get_filename()`, which returns the document's filename.
- `db_set_filename()`, which sets the document's filename to the character string which is passed to it.
- `db_get_sameflag()`, which returns the current value of the same-file flag.
- `db_set_sameflag()`, which sets the value of the same-file flag depending upon a comparison of the filename and the character string which is passed to it.
- `db_reset_time()`, which resets the time the user picked an area to N_A.
- `db_get_other_users_borders()`, which sends a message to the UI with the other user's borders (start and end points.)
- `db_set_buf()`, which sets the buffer indicated by the first parameter by reading from the file for as many characters as indicated by the second parameter.
- `db_user_alone()`, which sets all the buffers when the user is working alone by calling `db_set_buf()` appropriately.
- `db_users_together()`, which determines if the chosen node is being edited by the other user already. If it is, the UI is informed. If not, it sets all the buffers by calling `db_set_buf()` appropriately.
- `db_set_regions()`, which sets the time that the user chose the area and then calls either `db_user_alone()` or `db_users_together()`.
- `db_save_user_area()`, which saves the user's current working area by first storing the changes in the appropriate buffer and then adjusting the other buffer's

pointers, as well as the other user's starting index, if necessary. It then adjusts the length of the file and finally writes the new contents of the file to disk by writing each of the five buffers in order.

The next section contains a description of the problems which where identified in the DCWA prototype with respect to the database. An analysis of these problems lead to the design and implementation of the second version of the software.

4 Evaluation of the Prototype

The prototype served its purpose of helping the DCWA working group gain experience in this new area of Computer Science. Of the problems that were identified with the prototype as a whole, the following are some of its most obvious shortcomings with respect to the database [38] :

Cannot Add Empty Nodes to the Logical Structure
This feature is against the normal way of writing since it is said that a good writer first sketches the major blocks and the important points to be included in each block before he/she starts to write. Therefore, instead of only allowing a pre-existing text file to be divided up into nodes, the group should be allowed to create a structure without having typed a single character in the coauthored file.

The ability of the group to divide a pre-existing file into nodes will still be retained in the second version but it will be implemented differently. Basically, the abilities of modifying the logical view and editing a node will be completely separated. Therefore, the group should create the logical view and then edit the nodes, with the editing here consisting of copying and pasting from the pre-existing file rather than typing the information. From the point of view of the database, editing can be any combination of typing and/or copying/pasting.

Cannot Modify the Logical Structure Collaboratively
In order to avoid confusion, modification to the logical file structure was allowed when only one user was working on the file. The prototype was limited to two users and both the users could not modify the structure at the same time. In the second version, the users will be given the option of defining or redefining the logical view even while other group members are working. Any group member can modify the structure (i.e., divide nodes, merge nodes, etc.), except for nodes currently being edited.

Only One File Can Be Edited
Many examples exist where the group of coauthors work on multiple files at the same time, such as a project written in C that contains several modules, a makefile, several header files, etc. Therefore, DCWA should allow the group to work on more than one file at the same time. This problem is easily solved by simply expanding the role of the logical view to manage more than one file. The following restrictions will be retained, however :

- The contents of the files are represented by the leaf nodes and non-leaf nodes contain no information and are only used for the purpose of connecting the tree together logically.
- The leaf nodes may not represent overlapping parts of a file, but rather juxtaposed parts of a file.
- There should not be any part of a file which is not represented in the logical view.

And the following previous restriction has been slightly modified so that empty nodes are now permitted :

- The "granularity" of the various nodes may differ as defined by the group and may represent any number of characters from 0 to 32K.

Even when members of the same group are scattered geographically, they will still work by accessing the nodes of the same logical structure. That is, only one logical view per group per session is allowed. (But a user's *view* of it may be personalized – see below.) In other words, when more than one user is working at the same time, all the users have to open the same logical view file. Thus, the problems associated with many existing collaborative writing tools that create separate versions of a document (or parts of a document) are eliminated. Users can always access and edit the most up-to-date version of an unlocked node.

Files Must Be on the Same NFS
Along with the expansion of the role of the logical structure to incorporate as many files as the group needs, should also come the ability to store/retrieve those files in separate networked file servers (NSFs), depending upon the locations of the users. In a distributed environment it should be completely unnecessary that all the files of a group be located in the same file server. Furthermore, it is possible that the files will be distributed such that every group member holds one piece of the overall logical structure. The database will therefore have to store not only *file paths* but *machine names* as well. Users in different local area

networks should be able to work together in the DCWA across the Internet, collaborating on multiple documents per editing session. Thus a more realistic system where a group of geographically distributed people involved in multifile projects will be implemented.

The file containing a particular leaf node will be stored in the NFS of the machine which the group member is logged onto at the time when he/she creates the node. From time to time, nodes of a file may have to be downloaded to other machine's for editing. Saves will be stored on that machine but must eventually be committed to the master copy stored by the original NFS. This method is different from the SYBIL [33] and Office Work [28] systems, where a central server collects different and possibly conflicting versions from members and tries to resolve any conflicts.

User May Not Edit Multiple Nodes at the Same Time

Many times when a user makes a change in one node, a change will need to be made in another node. For example, changing the parameters of a C function found in one node will require the function prototype found in another node to be changed as well. Therefore, the second version should allow each user to read and/or edit as many nodes at the same time as desired. A user may still monopolize a node for as long as he/she wishes and a user may still watch edits as they happen at remote locations.

The User Interface Entangles with the Other Processes

One principle for user interface design is that the interface code must be kept separated from the application code. This principle is, unfortunately, not followed in the prototype and the "user interface" expands itself into both database and networking territories. Correcting this flaw so that the database is responsible for managing the logical view should improve the system's efficiency and its expandability.

The Restriction of Only Two Users per Group

The second version will remove this restriction and allow multiuser groups to simultaneously work together. A realistic maximum number of users per group will be established but it will be sufficiently large to prove that the DCWA permits "groupwork". However, managing the membership of the groups will not be a responsibility of the database but rather the daemon process, since the database process is not created until after a user determines which group he/she wishes to join. The database will, however, maintain a list of group members who are currently working, as well as the host machine each is working on. All members of a group will have unique member names, as enforced by the daemon.

The Restriction of Only One Information Medium

The second phase will also allow the group to coauthor ("codraw") graphics, as well as ASCII text. This expansion does not influence the design of the database, however. Rather the opposite, since the graphics files will need to be designed to fit within the existing logical structure where leaf nodes represent the actual contents (e.g., x, y coordinates of a line.)

The next section contains a description of the redesign of the DCWA database and some of the important choices that were made. The focus will be on the rationales behind these choices since it includes many novel approaches that deserve special attention.

5 Redesign of the Database

In the prototype, many of the database related problems were eliminated by the simplifying assumptions. However, the experience gained from implementing the prototype showed that a comprehensive, distributed algorithm for the database could become exceedingly complex. Therefore it was decided to implement the second version using the client-server model. All the database processes running on the group members' hosts will be considered "clients", except for one, which will be designated as the "server".

The database server is only a "server" in the sense that it serves other databases. Users, and more importantly, the database's sibling processes (CR, UI, and NA) will not be able to distinguish whether their DB process is a client or the server. That is, the server has all the functionality that a client does, as well as additional capabilities. It is thus a "central" database, though actually only a few things need to be requested from this central site. Most of the sibling processes' requests can be fulfilled locally and the main use of the server is to manage the edit locks of the nodes. It was decided that maintaining information consistency by managing the group's edits in this fashion would be easier than distributing this information and then running a "conflict-correction" algorithm. As before, a user will be able to select any leaf node to work on which is not currently locked by another group member.

In the discussion that follows, the focus is placed on the most important features that will be incorporated to enhance the database in the next phase of development. In general, the database process should provide services for backup and recovery, definition of node attributes, searching the logical view to define a partial view, sharing and conflict resolution, and finally, run-time modification of the logical structure. Some details of these capabilities are discussed below :

Backup and Recovery
The prototype's database process provided a full service for recording the local and remote

users' work areas and the logical structure of the document. However, only the most up-to-date version of the text is stored. Rollback, a common capability in all commercial database systems, was not supported. As such, users are not provided with the capability to undo a destructive action.

To implement backup and recovery, the database in the second phase will distinguish between "saves" and "commits". Saves of a node will be local and may be undone (and even redone) until the user has the contents of the node like he wants it. Commits, on the otherhand, are sent to the DB server and permanently replace the old version of the node. Only the content of that node will be replaced – the content of other nodes in the same physical file will not be affected by a commit, though the DB may need to update information about those nodes, such as their start and end points.

However, the ability to store multiple committed versions of a node to create a "history list" has been deferred to some future version of this project since this aspect needs to be discussed in more detail among the DCWA working group as how best to serve a group of users with such a feature. For example, when should the tool allow an old version to be deleted forever?

Node Attributes

The prototype only allowed the group to assign a label to each node in the logical structure, which is also the design choice in many other similar software tools. However, the information that can be carried by a label is quite limited and the second version will allow the group to assign several attributes to each node. For example, in an academic writing application, a node may correspond to a segment of text which mentions a certain chemical compound. The user may want to record this fact in the corresponding logical node as a quick reminder for the future. More importantly, the attributes can be used by the search facility (see below) to gain immediate insight to the *semantical* structure, as opposed to the organizational structure of the logical view. The objective is to combine both the "division of labor" and "shared mind" working styles discussed above into a hybrid style.

Therefore, in addition to the actual node content, each node of the logical structure will also contain attributes that describe the nature of the node. Some attributes will have their associated values assigned by the database, such as the node's id number and the time the contents of the node were last modified. Other attributes, such as the node's name (label) and any keywords, will have their associated values assigned by the group members. However, with two exceptions detailed later, there is no requirement for these types of attributes to have a value assigned by the group if they do not want or need it. Finally, the database will give the group the ability to define their own attribute/value pairs. In this case, the group not only assigns values but also specifies the names of the new attributes.

Search Facility

The documents, and their various sections, which the group is writing can become numerous and, thus, the logical structure would probably be a very complicated tree hierarchy. In order to help a user to locate a desired portion of the document quickly, search capabilities must be provided.

In group writing, especially in the case of "division of labor" writing, each writer only concentrates on his/her own part and repeated mentioning of the same point becomes possible. It has also been observed that collaborators need to know the details of a node written by another in order to use the node appropriately. If the node contains C program code, for example, the tediousness of the code often gives rise to the need for a reader to consult with the writer directly for clarification. In both these cases, an automatic search facility of the nodes' attributes would help a collaborator gain knowledge about what the others have written without having to actually read their work or ask them directly. It would also help a user quickly locate the node(s) he/she wants to edit.

The search facility allows a user to specify queries about the logical view. So far, few people have realized the definite need for a search facility with CSCW writing since in single user writing, the writer tends to know, and almost always remember, every detail of the document. In fact, search facilities are a common feature in most commercial databases and this feature should be available in a CSCW environment too. The use of node attributes and search techniques distinguishes the DCWA from other CSCW systems.

Definition of Views

The prototype made the entire logical structure available to both users following the principle of "no personal walls" [17] but normally not all users are interested in the entire structure until the final stages of the writing process. Therefore, a user should be able to specify a personal view based on his/her own interest and responsibility and this provision would be especially useful in narrowing the scope in a large node structure.

In the second version this capability will be implemented in conjunction with the search facility discussed above. Nodes which meet the query criteria will form a "hit list" and information about these nodes will be sent to the UI for display, replacing the entire logical view with only those nodes. If a remote user creates a new node (or modifies an old one) such that it also matches the criteria, it too will be sent to the UI for display as soon as the local DB is informed about it.

Storing these personal views between DCWA editing sessions will, however, be deferred to some future version since some questions regarding their use still remain. For example, how will their storage be related to the overall logical view's storage? How will the modification of the logical view during a DCWA editing session affect the personal view of a user who was not an active member during that session?

Sharing and Conflict Resolution

The ability to maintain the consistency of the file as it is being modified by the users is already a feature of the prototype and is handled by locking and unlocking the nodes of the logical structure. However, due to network delays it was possible (however slight since a LAN was used) that the two users selected the same node to edit. This possibility is greatly increased in the second version with the support of multiple, widely distributed users. To maintain consistency, a DB server, as discussed above, will be responsible for controlling the locks in the second version.

There have been a number of real-time conferencing tools that include conflict resolutions, such as MERMAID [23], SPIN [24], and the textual bulletin board in [16]. However, their methods are not quite as effective for resolving conflict arising in document composition.

Run-Time Modification of the Logical Structure

It is possible that the group creates a primitive logical structure at the very beginning of their project and will need to refine it as the writing process proceeds. In any event, a group member should be provided with the capability to define new nodes or delete unneeded nodes during the writing, without disturbing the other users.

The DB server will also be responsible for this conflict resolution and treats it just like edits. After the server locks a subtree of the logical view for modification, the user may make the necessary changes, save, undo, and redo just as if he/she were editing the content of a node. When the user commits the modification, the DB server will merge the existing logical view with the modified subtree, write the entire logical view into the file on disk, and then send the new subtree to all the DB clients. The term *subtree* was used since all descendants of the selected node are locked as well. Should the user need to modify the entire logical view, the root node should be selected.

Finally, this procedure may also be used to change the values of those node attributes which are assigned by the group. The user should lock the node for logical view modification, change the attribute's value and then commit, just as before.

The next section gives the complete specification of the DCWA's database for the second phase of development.

6 Specification of the Second Version's Database

Due to the expanded and complex nature of the database, as described above, the *Object-Oriented Paradigm* was used to separate the various capabilities into various classes. The database is a specialized, distributed program based on the client-server model implemented in C++ [39] running under 4.3BSD Unix. It is not a general purpose, commercial, distributed database and is, in fact, only one of five processes involved with DCWA as shown in Figure 6 (the daemon is not shown.) It is distributed among the group's hosts in two ways :

1. The logical view of the group is stored locally by each of the databases, thus making queries, reading, and saves (but not commits) local activities.
2. The DB Server automatically migrates to another host when the user of the machine where the DB Server resides decides to quit DCWA.

The first module of the prototype, `db.c`, will not need to be changed since pipes will still be used for communication between the local processes and the correct ends will still need to be closed, etc. The only modifications which were anticipated for the second module, `db_main.c`, were in the initialization section, since the *select* system call will still be used in the event-driven, message processing loop to determine when messages arrive on the pipes.

It was anticipated that new messages between the processes would be added, thus increasing the communication protocols, but `db_proc_msg.c` was to remain essentially the same as far as being a C switch statement with a case for each message. The emphasis of the second phase was, therefore, on the functionality formerly found in the last module, `db_log_view.c`.

6.1 Overview

The main purpose of this database is to allow any member of a group of users to select a "node", either for perusal or modification, so that a group of files may be cooperatively written by the group. A node is a part of a file and is defined by the group. In order to accomplish this task, the database needs the following from its sibling processes :

- Controller – whether or not this is the first database process created for the group for this particular working session.

- Network Access – to deliver error-free messages to any of the peer databases running on the other group member's hosts.
- User Interface – messages based on user actions with a unique code for each type of action and any associated data.

The database will, in turn, provide the following services to its sibling processes :

- Storage of the group's logical view (including node attributes) during its creation and/or modification.
- Searching the nodes by querying the attributes to present a personal view.
- Sharing and conflict resolution while text and/or graphic nodes are being edited.
- Backup and recovery while text and/or graphic nodes are being edited.

6.2 The User's Procedure

A user should begin his/her DCWA editing session as follows :

1. Opening a file containing a logical view or creating a new logical view.
2. Determining which node to select, and whether he/she wants to read the node, edit it, or modify the subtree of the logical view that has that node as its "root".

To determine which node to select, the user may rely on past experience (i.e., the user knows which node he/she wants) or may make a queried search of the available nodes to limit his/her view of the tree.

When a user gets a node for read-only, he may be informed (when it is received or perhaps later) that another user is currently editing the node's contents. The user should decide whether or not to watch the changes as they are being made. This capability may be "toggled" on and off as the user chooses until the remote editor finishes.

When a user gets a node for editing, the node becomes "locked" to other users who want to edit that node, though they may still read it, as described above. The same is true for users that lock subtrees of the logical view for modification.

During the editing process of a node the user has several options :

1. *Save* the current version as the top of a history stack of modified versions. (This stack is called the *undo stack* and the version of the node which was originally read in, is at the bottom of this stack.)

2. *Undo* by popping a version off the "undo" stack (and pushing it on the "redo" stack.) The version below the popped version will be sent to the UI for display.

3. *Redo* by popping a version off the "redo" stack (and pushing it back onto the "undo" stack.) This version will also be sent to the UI for display. If the "redo" stack is not empty when a user saves, all those old versions are deleted.

4. *Commit* the current version of the node to the group file. This also deletes any temporary files (see below) and the "undo" and "redo" stacks.

5. *Quit* without committing. This also deletes any temporary files and the "undo" and "redo" stacks.

When a user makes changes to the contents of the node and then quits without committing the changes, the database will perform the action as requested. That is, it is the UI's responsibility to make the usual checks concerning saving before the user releases the node.

A user may open (either for reading or editing) as many nodes as desired. For example, a user may have two nodes open for editing and have three other nodes open for reading. Also, the user may still make queries about the nodes even if a node has already been selected (i.e., the user does not have to release any nodes to make a query.)

It was decided that, unlike the prototype, holding the contents (of several versions) of the (possibly large) nodes in internal memory was inefficient and impractical. On the otherhand, since "save" usually implies to the user something more permanent than an area of storage in internal memory, it was decided to write the contents of the edited versions of a node to disk in the host machine's `/tmp` directory. The first design had the stacks holding filenames but a naming convention eliminated the need for this. Now only integers are needed to identify the top of the undo-stack, the redo-stack, and the number of elements on the stacks as a whole.

The naming convention for the temporary files is `/tmp/dcwa_#_#`, where the first pound sign is the node id and the second is the version number. For example, the third version of the fifth node would be saved in a file named `/tmp/dcwa_5_3`. Like the prototype, the second version of DCWA does not allow multiple users on the same host machine. If in a future version this policy is changed, the `/tmp` files of each user will need to be distinguished, possibly by adding a group id and a user id to the above naming convention.

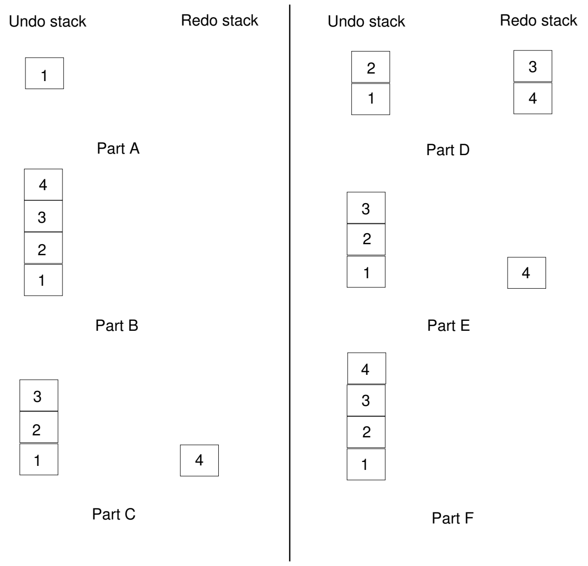


Figure 8: An Example using the "Undo" and "Redo" Stacks

An example of the undo and redo capabilities is shown in Figure 8, where the numbers are the version numbers.

- Part A shows the stacks after the user has selected a node for editing.
- Part B shows the stacks after the user has made changes to the contents of the node and issued saves until there are four versions.
- Part C shows the stacks after the user has decided that the last version was not what he/she wanted and has chosen to undo (step back to version 3).
- Part D shows the stacks after the user has decided to step back once more to have a look at version 2.
- Part E shows the stacks after the user has decided to step forward to version 3 by issuing a redo (step forward command).
- Part F shows the stacks after the user has made changes to the contents of the node and issued a save. The new version is saved as version 4, after any versions on the redo stack are deleted, in this case the old version called 4.

Finally, note that the use of the five working areas has been eliminated. Although an equation based on the number of users can be used to determine the number of areas needed, namely Number of Areas = (Number of Users * 2) + 1, this equation is based on a static system where there must always be that many users working together every session. The second phase of DCWA will allow the group members to enter and exit the group's session dynamically as they please. This is similar to "spontaneous conferences" in that not all users will join at the start of the conference. The system should allow users to join the group effort at any time, as well as allowing existing members to leave at any time [12].

Another observation related to this is the fact that only the part of a physical file occurring after a node to be committed needs to be read into a temporary buffer. In contrast, the prototype read in the text before the node only to write it back out again, which was inefficient.

6.3 Features

6.3.1 Logical View

1. The group members should be able to define a logical structure without any text (or graphic object, etc.) in the nodes, unlike the prototype where the group must start with a pre-existing text file and then break it up. That is, they may know (a first draft of) the logical structure that they want without having typed a single character yet. If the group does not want to get too complicated the logical view can be as simple as one root node followed by as many child nodes as necessary, for example :

```
Paper __
        |- Abstract
        |- Intro
        |- Point 1
        |- Point 2
        |- Point 3
           ...
        |- Conclusion
        |- Notes
        |- Bibliography
        |- Appendix I
        |- Appendix II
```

Another logical view (presented graphically) is seen in Figure 9, where the term "Sections" is used rather than "Pages" because an integer number of paragraphs need not fit on a physical (printed) page (i.e., part of the paragraph might be on one page and the rest of it would be on the next page.) In a tree structure, however, there can only be an integer number of child nodes per non-leaf node. If a group wants to divide a paragraph into two parts (one part on one page and one part on the next) that is fine but there is no way in the current design to guarantee that the parts will remain that way as the group edits those nodes or any other nodes in the same physical file. (That is, something similar to ODA's "layout structure" has not yet been incorporated into DCWA.) Finally, unlike the structure of note taking where having a subtopic 1 requires a subtopic 2, it is permissible to have a non-leaf node with only one child node (either leaf or non-leaf).

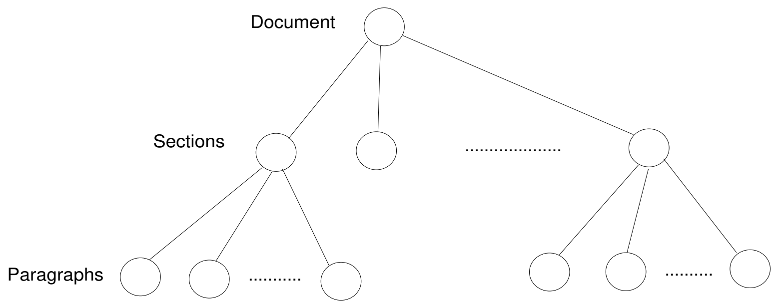


Figure 9: Graphical Example of a Logical Structure

The terms that the group uses to name nodes when defining the logical structure is unimportant to the database. A node's name is just one of many attributes that it maintains for each node. (See the section below on node attributes.) The database identifies and differentiates nodes by a node id which is maintained internally by the database itself and not assigned (perhaps erroneously) by the group. Therefore, the prototype's restrictions placed upon node names have been eliminated :

(a) "@" is nothing special and may be used as a node name, if it means something to the group.

(b) Node names do not need to be unique, if it is not confusing to the group.

Also, from the database's point of view, empty node names would be acceptable but it was decided to prevent the group from assigning an empty node name since these node names are the main way to distinguish the nodes as seen in the UI's presentation of the logical view. This is the first of two restrictions placed on the user-assigned values of a node's attributes. However, the group may use one or more spaces as a node name if they really want the node name to look empty. This policy concerning spaces may change in future versions, however.

Another example of a logical view is seen in Figure 10, which represents a possible structure for a C program file.

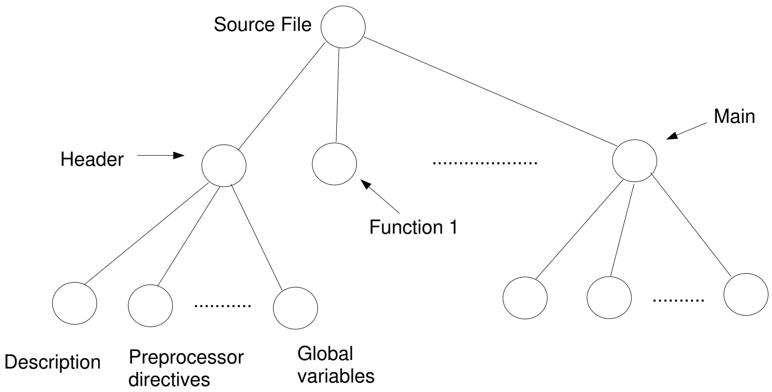


Figure 10: Another Graphical Example of a Logical Structure

The group should be able to manipulate the logical structure in two different ways :

(a) Start with a default structure of 1 node. Define a structure without specifying the start and end points for each leaf node. Save the structure.

(b) Start with an existing structure, lock the node that is the parent of the nodes that are going to be modified. Modify that subtree of the logical structure dynamically while other user's are reading, editing, or modifying (another subtree of) the logical view. Save the new structure.

The first method is local to the user's database. The user informs the database that he wishes to create a new logical view. The user then uses the functionality provided by the UI to create the tree and fill in the node attributes. The user may save, undo, and redo the changes until the logical view is what he/she wants. When the user wishes to commit the new logical view, the UI will ask for a filename. If the file already exists, the UI should ask if the user is sure that he/she wants to overwrite the existing file or not. (The database does not check for this and will simply overwrite the existing file.) The UI should then send the logical view to the local database, which will store it in the specified file after adding some other attributes to each node. The user may also quit this logical view modification process without committing the new logical view.

The second method requires informing the database server, which checks if it will allow the user to modify (a subtree of) the logical view. Possible reasons why it would not give permission include :

- The user does not have permission to modify the logical view – see the node access policy below.
- Another user is editing the node or one of the node's descendants.
- Another user is already modifying a subtree of the logical view which intersects with the requested subtree.

If the user does get permission, he/she will then use the functionality provided by the UI to modify the tree. The user may save, undo, and redo the changes until the logical view is what he/she wants. All these operations are local as before. The user should then either commit the logical view or quit the process without committing. If he/she decides to commit, the UI need not ask for a filename but will send the information to the local database as before, which again adds some other attributes to each node. The local database will then send the new subtree to the DB Server, which will merge it with the old logical view to form an updated logical view. The server will then store the logical view on disk and broadcast the new subtree to all the DB Clients. The DB Clients will also have the capability to merge the new subtree with the old logical view but they will not store the result on disk.

Note that the prototype's method of opening an existing (text) file and dividing it up is still available but the method has been somewhat modified. Namely, the group should define the structure as per one or two above and then users should open the appropriate nodes for editing, not logical view modification. They should then open the pre-existing file and copy and paste the appropriate contents into each node. The user may save, undo, redo, and finally commit the changes just as if he/she were typing in the node contents from scratch. That is, the database does not care how the information gets into a node. It might be typed in by the user, copied and pasted, or any combination of both. To the database, it is all just editing.

Finally, this method allows the group to insert (or append) an existing file into (at the end of) another existing file, which was impossible with the prototype. The method is the same as before – create the nodes, copy and paste the information, and then commit.

2. In the prototype, the root node represented one file, which presents a problem when other media beyond text are to be added. For example, how will graphical objects be incorporated into the file? See the following :

```
Node 0 (File) ___
                 |
                 | - Node 1 (text)
                 |
                 | - Node 2 (graphics) ???
                 |
                 | - Node 3 (text)
```

One solution is to allow Node 0 to represent more than one file :

```
Node 0 (Files) __
                 |
                 | - Node 1 (text file)
                 |
                 | - Node 2 (graphics file)
                 |
                 | - Node 3 (text file)
```

But this would have probably broken the user's train of thought to have to break up text before and after the graphics figure into separate files. The solution used for the second phase will be to let any leaf node be a "file pointer" (i.e., the logical structure is independent of files) :

```
Node 0 (Files) __
                 |
                 | - Node 1 (text file 1 : 0 - 100)
                 |
                 | - Node 2 (graphics file A : 0 - 56)
                 |
                 | - Node 3 (text file 1 : 101 - 300)
                 |
                 | - Node 4 (graphics file A : 57 - 99)
```

Here, Nodes 1 and 3 are next to each other in text file 1 (as shown by the start and end points) and Nodes 2 and 4 are next to each other in graphics file A. Note that files start at index 0, just like C arrays.

3. Node ids will be assigned by the database in a depth first manner. For example, for the following structure, the ids are as shown :

```
0 ____
      |
      | - 1 ____
      |         | - 2
      |         |
      |         | - 3
      |         |
      |         | - 4
      |
      | - 5 ____
      |         | - 6
      |         |
      |         | - 7
      |
      | - 8 ____
                | - 9 ____
                |         | - 10
                |         |
                |         | - 11
                |
                | - 12
                |
                | - 13
                |
                | - 14 ___
                          | - 15
                          |
                          | - 16
```

Whenever this structure needs to be passed to another process, the nodes will be enclosed in a "Bison-like" notation as follows :

```
0 %{1 %{2%%3%%4%} 5 %{6%%7%} 8 %{9 %{10%%11%} 12%%13%%14 %{15%%16%}%}%}
```

The above notation only showed the node ids but actually any or all of the attributes may be sent as necessary. Attributes will be separated by `\n` (not used above to save space) and leaf nodes will be separated by `%%`, such that every node (except the root node) will have at least one of `%{`, `%}`, and `%%` on either side of it. These special symbols are not "keywords" and may appear in the values of an attribute if necessary. They are for the convenience of someone looking at the logical view file to easily see the separate nodes. However, since `\n` is used to separate attributes, it is not allowed in any field. Also, spaces are used to separate keywords, user-defined attributes, user-defined attribute values, and user names in the access lists, so they may not be a part of the values for those attributes. (See the section on node attributes below.)

4. The leaf nodes representing parts of a physical file may be distributed throughout the logical view in any way. For example, assuming filenames of File1 and File2 :

```
0 root -
        |
        |- 1 partA -
        |           |
        |           |- 2 File1
        |           |
        |           |- 3 File2
        |
        |- 4 partB -
                    |
                    |- 5 File1
                    |
                    |- 6 File2
                    |
                    |- 7 File1
```

Since the database maintains the start and end points internally, there is no possibility of overlap, as long as the files are edited from within DCWA. Empty nodes have start and end points of -1. Non-empty nodes with lower node ids have start and end points less than nodes with higher node ids. The following is an extended example :

After a user creates the logical view, all leaf nodes are empty :

```
0 root -
        |
        |- 1 partA -
        |            |
        |            |- 2 File1 -1 -1
        |            |
        |            |- 3 File2 -1 -1
        |
        |- 4 partB -
                     |
                     |- 5 File1 -1 -1
                     |
                     |- 6 File2 -1 -1
                     |
                     |- 7 File1 -1 -1
```

The nodes of a file may then be edited in any order, thus supporting non-sequential authoring. For example, File1's associated nodes are 2, 5, and 7. The possible sequences of editing are :

```
1) 2 5 7                4) 5 7 2
2) 2 7 5                5) 7 2 5
3) 5 2 7                6) 7 5 2
```

Cases 1 and 5 are shown in detail below. (The other cases would be similar.) For this example, the node sizes have been arbitrarily chosen as :

```
Node 2 --  500 bytes
Node 5 -- 1001 bytes
Node 7 -- 1201 bytes
```

Case 1) A user edits node 2 and commits :

```
0 root -
        |
        |- 1 partA -
        |           |
        |           |- 2 File1 0 499
        |           |
        |           |- 3 File2 -1 -1
        |
        |- 4 partB -
                    |
                    |- 5 File1 -1 -1
                    |
                    |- 6 File2 -1 -1
                    |
                    |- 7 File1 -1 -1
```

A user edits node 5 and commits :

```
0 root -
        |
        |- 1 partA -
        |           |
        |           |- 2 File1 0 499
        |           |
        |           |- 3 File2 -1 -1
        |
        |- 4 partB -
                    |
                    |- 5 File1 500 1500
                    |
                    |- 6 File2 -1 -1
                    |
                    |- 7 File1 -1 -1
```

A user edits node 7 and commits :

```
0 root -
        |
        |- 1 partA -
        |           |
        |           |- 2 File1 0 499
        |           |
        |           |- 3 File2 -1 -1
        |
        |- 4 partB -
                    |
                    |- 5 File1 500 1500
                    |
                    |- 6 File2 -1 -1
                    |
                    |- 7 File1 1501 2701
```

Case 5) A user edits node 7 and commits:

```
0 root -
        |
        |- 1 partA -
        |           |
        |           |- 2 File1 -1 -1
        |           |
        |           |- 3 File2 -1 -1
        |
        |- 4 partB -
                    |
                    |- 5 File1 -1 -1
                    |
                    |- 6 File2 -1 -1
                    |
                    |- 7 File1 0 1200
```

A user edits node 2 and commits :

```
0 root -
        |
        |- 1 partA -
        |          |
        |          |- 2 File1 0 499
        |          |
        |          |- 3 File2 -1 -1
        |
        |- 4 partB -
                   |
                   |- 5 File1 -1 -1
                   |
                   |- 6 File2 -1 -1
                   |
                   |- 7 File1 500 1700
```

A user edits node 5 and commits :

```
0 root -
        |
        |- 1 partA -
        |          |
        |          |- 2 File1 0 499
        |          |
        |          |- 3 File2 -1 -1
        |
        |- 4 partB -
                   |
                   |- 5 File1 500 1500
                   |
                   |- 6 File2 -1 -1
                   |
                   |- 7 File1 1501 2701
```

The result of the above is simply that the start and end points are the same after all three nodes have been edited and are independent of the order in which the group edited the nodes.

6.3.2 Node Attributes

The following is the list of attributes that are maintained for each node, as well as an indication of the type of information each will hold :

```
Node id : An integer.
Name : A character string label for the node.
Node level : One of Root, Intermediate, or Leaf.
Creation time : A number of type time_t.
Creator : A character string holding a username.
Last change time : A number of type time_t.
Last Editor : A character string holding a username.
Topic : A character string.
Description : A character string.
Keywords : A character string with each keyword separated by a space.
User Defined Attributes : A character string with each attribute
                          separated by a space.
User Defined Attribute Values : A character string with each value
                                separated by a space.
Read access list : A character string with each username
                   separated by a space -- see the node access policy below.
Edit access list : A character string with each username
                   separated by a space.
Logical View Modification access list : A character string with each
                                        username separated by a space.
Local user reading : A boolean.
Current editor : The editor's database id number -- see the id policy below.
Lock mode : One of Edit or Logical View Modification.
Media type : One of Text, Graphics, Image, Audio, or Video if a leaf node,
             or Logical if not a leaf node.
             Only Logical, Text or Graphics should be used in the second
             version.
```

```
Filename : A character string with the machine name and full path name where
           the file is stored if a leaf node or NULL if a non-leaf node.
Start point : An integer.
End point : An integer.
Number of children : An integer.
```

When the logical view is created or modified, the UI will supply certain attributes' values, as obtained from the user, separated by a newline character, `\n`. [] means optional but if empty, the UI still sends `\n` to separate the fields. The three access lists are conditionally optional, as explained below, and are marked by [[]]. In short, a node's access lists may be empty, in which case it inherits its parent's access lists. However, if all the access lists of all the nodes are empty, no member of the group will be able to do anything. The UI should also send the node separators – `%{`, `%%`, and `%}` – where appropriate. The following node attributes' values are supplied by the user :

```
Name
[Topic]
[Description]
[Keywords]
[User Defined Attributes]
[User Defined Attribute Values]
[[Read access list]]
[[Edit access list]]
[[Logical View Modification access list]]
Media type
Filename
```

The UI supplies the Number of Children attribute's value based on the tree the user created. When the user commits the changes made to the logical view, the DB adds these attributes to each node :

```
Node Id
Node level
Creation time
Creator
Last change time
```

Last Editor
Start point
End point

The start and end points are initially -1 until a user actually puts something into the leaf node. The Local user reading, Current editor, and Lock mode attributes are only used as the group collaborates and are not stored in the logical view's file on disk.

Finally, there are relationships between some of the attributes such that some of them may not have been needed (i.e., the values may have been calculated) but for the second phase they were all left in so as to facilitate the logical view validation process. The relationships are as follows, where N_A = -1 :

	Node Level		
Media Type	ROOT	INTERMEDIATE	LEAF
LOGICAL	node_id = 0 num_children != 0 filename = "" start & end = N_A	node_id != 0 num_children != 0 filename = "" start & end = N_A	Not allowed
OTHER MEDIA	Not allowed	Not allowed	node_id != 0 num_children = 0 filename != "" start & end = or != N_A

Two implications of the above are :

1) If a single user wants to edit a file in the DCWA in a traditional manner (i.e., without logically breaking it up into nodes), the logical view must have at least a Root node above the node representing the file in the tree. For example :

```
0 root -                    or      0 root -
        |                                   |
        |- 1 File1 0 9999                   |- 1 dummy -
                                                        |
                                                        |- 2 File1 0 9999
```

2) At the time of committing a logical view, a Logical node cannot be a Leaf node, even if children are to be added under it during a later session. Either dummy leaf node(s) should be created under it or it should be removed until that session.

6.3.3 Node Access Policy

When a node is selected for reading, editing or logical view modification, the appropriate access list is checked to determine if the user has permission to perform that operation on that node or not. It is assumed that any user who can edit a node can also read it, so the username need not appear on both lists. Similarly, a user that has permission to modify the subtree of the logical view starting at the given node also has permission to edit it and read it.

There is thus a hierarchy among the access lists of a given node. Any of the lists may also be empty for a given node, in which case the node will inherit its parent node's corresponding access list. If the parent node's access list is also empty, *its* parent node's access list will be consulted, etc.

If a group deems it unnecessary to specify access lists for every node, they may simply put all the user's names in the logical view modification access list of the root node of the logical view. Doing this will allow all members of the group to read and edit every leaf node, as well as modify every part of the logical view. Specifying the usernames on this list is the minimum requirement for the access lists and is the other of two restrictions placed on the user-specified attributes.

If, on the otherhand, the group feels it necessary to specify the various permissions for each individual node, the database provides that capability as well. Most groups will probably fall somewhere in between – specifying some of the lists and letting others take on the inherited values.

6.3.4 Queries

A user may make a queried search of the logical structure to limit his/her personal view of the tree. There are two types of queries, new and revised. A new query searches the group's entire logical view for matching nodes but a revised query only searches the nodes that matched the previous query (either new or revised.) In either case, a "hit list" (perhaps empty) of matching nodes will be returned to the UI and the user may then select a node from among those in the hit list to read or edit or the user make another query to further limit the view. If a query does not produce any nodes that the user is interested in, he/she may start over by making a new query, which again searches the entire logical view.

1. New Query :

   A message is sent with the data of the form :

   ```
   DISPLAY : List of attributes to display, each separated by a comma\n
   FOR : query criterion of the form : Attribute Operator Test-Value\n
   ```

   For example,

   ```
   DISPLAY : Node Name,Creator,Description,User Attribute-Value Pairs
   FOR : Node Id > 3
   ```

   For the attributes specified in the "display part", the associated values are returned to the UI separated by `\n`. In addition, the node id is always sent to the UI as the first attribute value so that the UI may identify which nodes are in the hit list in case further communication with the DB concerning them is necessary, for example if the user selects one of the nodes to edit. The other attributes are sent to the UI in the order the user specified in the "display part" of the query, not their order as stored in the node itself. Finally, if more than one node meets the query criteria, each node will be separated by `%%\n`. Note that the nodes in the hit list/personal view are no longer arranged in a tree structure since such an organizational structure does not exist. That is, they are simply a list of nodes which meet the query criteria.

2. Revised Query :

   The user may make a revised query based on what was obtained from the previous query. A message is sent and the data is of the same form as above, however :

- The "display part" may be different than it was in the previous query, with either more or less attributes to display. If an attribute was displayed in the previous query it must be listed again if the user still wants that attribute's value displayed.
- The "for part" will normally be different. If it is different, the resulting hit list is ANDed to the previous hit list to form the new hit list. If it is not different, the hit list will remain the same.

Most attributes may use `==`, `!=`, `>`, `>=`, `<`, and `<=` as operators in the "for part" but some attributes have a limited number of operators that can be used. For example, Node Level, Lock Mode, and Media Type only use `==` and `!=`. The keywords, user-defined attributes and three access lists attributes also use only `==` and `!=` but here `==` means "is a member of" and `!=` means "is not a member of". For example, a criterion of "keywords == GUI" would search for nodes with "GUI" in its list of keywords, not for nodes with "GUI" as the only keyword.

Some "attributes" not specifically stored by the database may also be displayed and/or used in the criterion, because the database can calculate the value. Namely, the day, date and time may be separated and the size of a leaf node (calculated from the start and end points) may be requested. The Current Editor and Lock Mode attributes may be used in queries but not the Local Reader attribute, since a user should know whether or not he/she is reading a node.

Finally, the Description attribute may be displayed but it cannot be used as the criterion, as in Description == "....", since it is highly unlikely that a user will type in the (possibly long) description exactly as it is stored in the node.

6.3.5 Semantic Network

Many existing CSCW tools only allow users to associate a label with each node of the organizational structure, however, the information that can be carried by one label is quite limited. Therefore, the node attributes as described above were designed and implemented. Once these attributes are assigned values, a search of the logical structure may be made to limit the tree to only those nodes which the user is interested in. However, these attributes may also be used to develop a semantic network on top of the organizational structure of the logical view.

For example, a group of chemists writing a paper may have a node which corresponds to a segment of text that mentions a certain chemical compound C_1. For future reference, the user may assign the following attribute/value pair to the corresponding node :

$$\text{CHEMICAL\_COMPOUND } C_1$$

In addition to assigning attribute/value pairs to a node, the database also helps the user relate all these attributes in an *inheritance* structure. For example, the user knows that "CHEMICAL_COMPOUND" is a subconcept of "CHEMICAL." Therefore, the user may also want to add the following attribute/value pair to the node :

CHEMICAL CHEMICAL_COMPOUND

Clearly, the attribute/value pair can be used to encode the "is-a" semantic, but it may also be used for encoding the domain relationship among concepts by listing a user-defined attribute as the value of another user-defined attribute. These two types of semantics are believed to be the fundamental elements for all possible semantic relations [40].

By associating each node with the user-specified attribute/value pairs, a semantic network is overlaid upon the organizational tree as seen in Figure 11. By using the semantic network and the search facility, users may compose queries to locate all nodes that satisfy a certain condition. The result of the query may also be used for a finer search until the needed information is found. This functionality is similar to what is provided by the Information Lens [14] and the Object Lens [15].

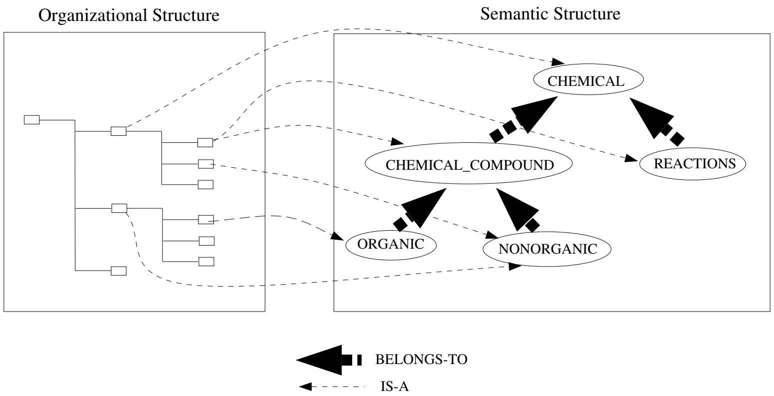


Figure 11: An Example Semantic Network

In distributed, group writing, users' awareness of the entire structure may hinder the "division of labor" but having a database containing information about the semantical structure will greatly reduce the users' burden when they need to quickly perceive what the others have written.

Finally, it is interesting to note that the hypertext solution in Quilt [27], ForComment [28], rIBIS [29], and SEPIA [20] is, in fact, a primitive form of overlaying a semantic net over the organizational structure.

6.3.6 Id Policy

To distinguish between the various peer databases of a group, an id policy was established. The DB Server keeps an array of booleans of size MAX_USERS. If an element of the array has a value of 0, it indicates that that id has not yet been assigned to any database in the group. Likewise, a value of 1 indicates that it has been assigned. These ids are valid only during a working session. The next time a user starts DCWA, his database will probably not get the same id. If it does, it is only a coincidence and not a permanent assignment. Finally, these ids are only used internally among the databases themselves. Their sibling processes, hosts, and users do not use them, though for convenience if a DB has an id of 4, its sibling UI, NA and CR will have an (implicit) id of 4. The host machine and user may also be referred to as having an (implicit) id of 4.

These ids are also only relevant within one group. There may be several groups running DCWA and all of them could have a database whose assigned id is 3, for instance. If a user runs two instances of DCWA, one for one group and another for another group, then it would only be coincidence if the two databases had the same id within both groups.

However, the database will need to keep a "translation table" which stores the username and hostname in case one of its sibling processes needs that information. For example, if a user queries the database about nodes currently being edited by A_Smith, the UI can not translate A_Smith to some corresponding id which it does not even know about. The database will do the translation and then determine which nodes, if any, are being edited. Likewise, the NA process does not need to know the DB's id. If a DB needs to send a message to its peer DB that has an id of 5, it sends the message to its sibling NA process instead. The NA process broadcasts the message to all the other NA processes in the group and then each of them passes the message to their respective sibling DB. The DB with an id of 5 will get the message and do something based upon it but all the other DBs will ignore the message.

A maximum of 8 users, numbered 0 to 7, is assumed for the following examples. The id array which the server keeps will be initialized to all zeros, except for element 0, which represents the server itself :

```
0 0 0 0 0 0 0 1        value
7 6 5 4 3 2 1 0        element (id) number
```

When another user in the same group starts DCWA, the id assigned will be the lowest in the array which has a value of 0. In the case of the second user to join the group session, the id assigned will therefore be 1 :

```
0 0 0 0 0 0 1 1
7 6 5 4 3 2 1 0
```

The next DB would be assigned an id of 2, then 3, etc. When a user decides to quit DCWA, the appropriate element will be changed back to 0.

As an example, suppose that five users are currently running DCWA. The array will look like :

```
0 0 0 1 1 1 1 1
7 6 5 4 3 2 1 0
```

Now user 2 decides to quit and so the array looks like :

```
0 0 0 1 1 0 1 1
7 6 5 4 3 2 1 0
```

If a new user were to start DCWA right now, the id assigned would be 2, not 5, since 2 is the lowest element that has a value of 0.

But assume that user 1 decides to quit before the new user joins in the working session. The array will then look like :

```
0 0 0 1 1 0 0 1
7 6 5 4 3 2 1 0
```

When the new user starts DCWA, the id assigned would now be 1, not 2, since 1 is the lowest element that has a value of 0. When the user does join, the array will look like this again :

```
0 0 0 1 1 0 1 1
7 6 5 4 3 2 1 0
```

When a user is the first to start a working session, her DB is the DB Server for the group. When he/she decides to quit, the server must migrate unless there are not any other members of the group currently working. The DB, which is not the current server, whose id is the lowest is chosen to become the new server and messages are sent to instantiate a server on the new host.

Continuing with the example from above, the DB whose id is 1 would become the server :

```
0 0 0 1 1 0 1 0
7 6 5 4 3 2 1 0
```

Had the new user in the example above started DCWA after the user quit whose DB was the old server, the DB whose id is 3 would have been chosen instead.

Finally, if another new user starts DCWA now, the assigned id would be 0, not 2, since 0 is the lowest element that has a value of 0. The result is shown below but in this case the server does not need to change back to host 0. It can stay on host 1, until that user decides to quit, in which case the server will again migrate to whatever active DB has the lowest id at that time.

```
0 0 0 1 1 0 1 1
7 6 5 4 3 2 1 0
```

The database knows whether or not it is the first DB created for the group's current session because the DCWA daemon sets a flag. Therefore, if two users of the same group start DCWA at nearly the same time, two DB Servers will not be instantiated. This first DB Server for the group's current session will always assign itself an id of 0 but, as seen above, the server may eventually be a DB with any id. The reason for this policy was to assign an id to a DB throughout a working session instead of confusingly changing the ids around as users joined and/or quit from that session. Thus, the server may have any id and is not permanently assigned an id of 0 as it migrates around.

Originally, the migration of the DB Server was designed to be as follows. The server would receive a quit message from the UI and would search for a client in the same group to migrate to. If one could not be found, it would quit. If one could be found, it would send a message to that client asking if it could be the new DB Server.

If the chosen client could not become the new server (because it had already sent out a quit request message), the server would search for another possible client to migrate to and again would either quit if one could not be found or send a message if one could be found.

If the client agreed to become the server, the old server would send it any relevant info and wait until it heard back from the new server. In the mean time, the old server would store any messages it received on a queue and would send those messages to the new server when it heard back from the new server.

However, this caused some messages to get lost and/or sent out of order because the NA arranges the hosts of a group in a logical ring. For example, after a DB Client's NA puts a message "out on the ring" assume it comes to the new server first. The new server would ignore the message because it has not received the info from the old server yet and knows the old server is maintaining the queue. Meanwhile, assume the old server's NA puts the needed info and queued messages out on the ring. The old server would then quit and when the new server's NA passes the client's message on, it would never arrive at the old server and so would never be put onto the queue. Eventually, the client's NA receives the message after it has gone around the ring once and deletes it. Thus, the DB Server never received the message.

A slight change of perspective solved the problem, though. The new design is the same as described above except that the new server maintains the queue of messages sent by the DB Clients. When the new server receives the info from the old server, it does not need to send a message back to the old server but simply begins processing the queued messages one at a time. When it finishes with the queued messages, it is "open for business" as the DB Server.

6.3.7 Groups

Figure 12 shows four DCWA groups, represented by the large circles. As far as the database processes in one group are concerned, the other groups do not exist. The number of members per group in the second phase of DCWA will be from 1 to 8. The other symbols in the figure are explained below :

- A small circle represents an active user (or database process).

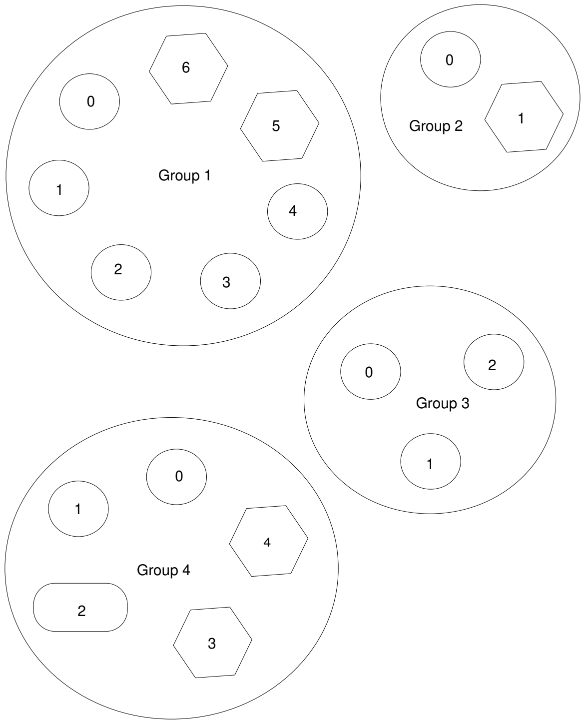


Figure 12: Four separate DCWA groups

- A hexagon represents a "slot" for a user to join the group.
- A rounded rectangle represents a database process which is "starting up". Either it has not yet informed the DB Server of its existence or has not yet received its id from the DB Server.
- The numbers are the ids as explained above in the id policy.

As seen in Figure 12, Group 1 has seven members (ids 0 - 6), five of which are currently working on their collaborated documents.

Group 2 has two members but only one of them is currently working on their documents.

Group 3 has three members and all three are currently working on their documents.

Group 4 has five members, two of which are currently working on their documents and a third is in the process of starting up and will soon be collaborating with them.

Figure 13 shows that users may belong to more than one group at a time by starting multiple versions of DCWA and then specifying different groups to join. Note that the databases will have different ids, unless by coincidence the same id is assigned.

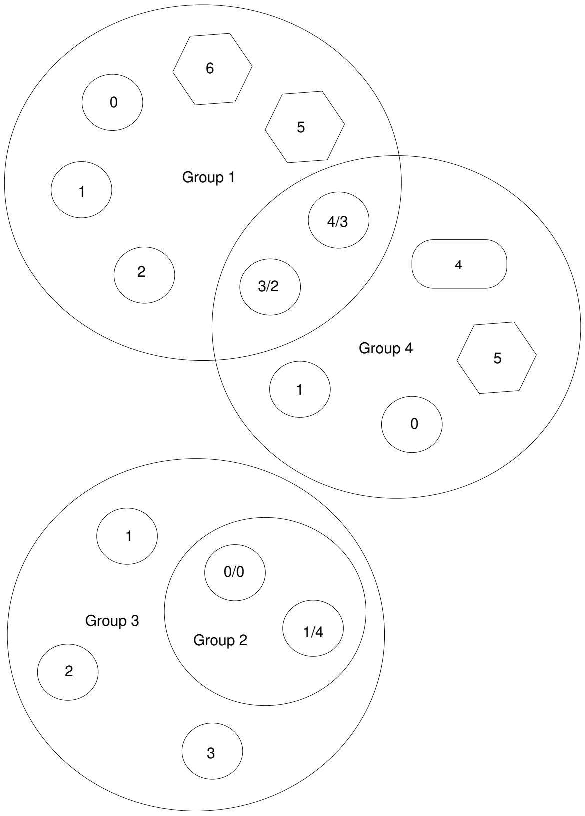


Figure 13: Intersection and Union of Groups

As seen in Figure 13, two users are active members of both group 1 and group 4. One has an id of 3 in group 1 and an id of 2 in group 4. The other user has an id of 4 in group 1 and an id of 3 in group 4.

Two users are also active members of groups 2 and 3. One has an id of 0 in both groups but the other has an id of 1 in group 2 and an id of 4 in group 3.

6.3.8 Client and Server

The `db_client` class is used to create the object which will oversee the DCWA database activities on a user's host. Some of its methods (services) include : Process Query, Select a node, Save a node, Commit a node, Undo, and Redo as described above. (See the section on classes later for full details.)

When it is created, it sends a message to the DB Server informing the server that it is "alive". The DB Server will send back a unique id to identify the client in all further communication, as explained in the id policy above.

When the user wishes to open a logical view file, the UI should inform the database. That database will inform the DB Server, unless of course it is the DB Server too, which will then send back the group's logical view, including the node attributes. The DB Client will store this information and most other operations may be made locally. Note that, unlike the prototype, the user should open the logical view file not any of the files represented in its structure.

The `db_server` class is used to create the "central" database object which gets created and runs on the machine of the user who first starts a DCWA working session for a group. Note that the DB Client and DB Server are not two separate processes running on the same machine. There is only one DB process per host which is running DCWA and, within each group, one of the DB processes instantiates a DB Server object and all the others instantiate a DB Client object.

However, this class is a derived class of the `db_client` class, since it must include all that functionality as well for local requests. Thus some of the member functions of the `db_client` class are declared *virtual* so that they may be overridden in the `db_server` class. For example, when a member function of the `db_client` class must send a message to the DB Server via the NA, the overridden member function defined in the `db_server` class will act directly.

Thus, the user whose database is the DB Server will have a certain advantage over other users in that he will not experience any network delay. However, there will be a slight disadvantage since the server must process messages from the other databases (i.e., spend CPU cycles not devoted to local messages.) However, this should not overburden the server's host machine since messages to it will most likely occur very infrequently since most of the time the users will be querying, reading, and/or editing, which are actions performed by each user's local database.

The DB Server is responsible for making sure that only one user opens a node for editing at a time. When a user selects a node for editing and gets it, the DB Server will inform all the other databases of the "lock", so that the corresponding UI can display the node in the appropriate color. If a user tries to select the same node, due to a network delay, the DB Server will catch the request and notify the local database of the lock. Even though the node is locked, other users may read the content of the node and/or watch the modification process, as described above.

When the user who started the session decides to quit, the server will `migrate` to another user's machine, unless, of course, there are not any other users in the group currently running DCWA. The choice of host was explained above in the section on the id policy.

Finally, note that in some cases, when a message code is received, a member function is called which does nothing but pass the message on (after changing the associated code), either to the UI or to the DB Server, for example. This message passing could have been done without creating a new member function but they were created :

- So that the message-processing switch looks "symmetric". That is, a member function is called whenever a valid code is received. What happens inside that member function should not be important to the "case" statement that dispatched it. This is one example of *information hiding* used by the database.
- So that future expansions may be easily incorporated when the associated message code is received. That is, any additional processing may be easily added to a member function which already exists.

The next section explains the classes of the second version's database in detail.

7 Classes

The database was designed using the object-oriented paradigm and implemented in `C++`. Each class that was designed and implemented is now explained in detail. (N_A was defined as -1.) Though some of the classes are similar, such as DB_Queue, LongString, Update_LV_List, and ValidFile_List, it was felt that they were different enough not to use a template class, since such a template would require all the various member functions that each of these separate classes provide for interacting (differently) with the cells in their respective lists. It would therefore have probably been very confusing to a reader of the code as to what the overall purpose of the template was.

7.1 DB_Stack

This class is used to store the various versions of a user's edits until he/she commits the changes. The versions are stored in the host machine's /tmp directory and a naming convention (using numbers) is used to distinguish them. Therefore, the "stack" itself needs only store the important numbers.

The DB_Stack class has the following private data members :

1. An integer called `node_id`, which indicates for which node this stack has been instantiated, since a user may edit more than one node at a time. The node id is also used in the naming convention.

2. An integer called `undo_top`, which indicates which version is currently on top of the "undo" stack.

3. An integer called `redo_top`, which indicates which version is currently on top of the "redo" stack.

4. An integer called `num_elements`, which indicates how many versions are in the /tmp directory for this node.

The DB_Stack class has the following public member functions :

1. A Constructor which takes an integer parameter indicating the `node_id`. Another reason why this id is stored is so that an object of class DB_Node does not have to pass in the `node_id` to the other DB_Stack member functions, especially the destructor, which cannot have any arguments. After setting `node_id` to the indicated parameter, this function sets `undo_top` and `num_elements` to 0 and `redo_top` to N_A.

2. A Constructor which takes four integer parameters indicating the `node_id`, `undo_top`, `redo_top`, and `num_elements`, respectively. Each data member is set to the associated parameter. This constructor is for when a DB Client becomes the new DB Server and the stacks must be "passed" to the new DB's logical view.

3. A Destructor which deletes any /tmp files created by this instantiation of DB_Stack.

4. `get_undo_top()`, which returns the value of `undo_top`.

5. `get_redo_top()`, which returns the value of `redo_top`.

6. `get_num_elements()`, which returns the value of `num_elements`.

7. `undo()`, which takes an integer parameter indicating a database id number. It opens the file indicated by `undo_top` and reads in its content. It then sends the content to the UI (specifying the DB id number as the sender.) Finally, it sets `redo_top` to `undo_top` then decrements `undo_top` by one, unless it is already one. Version 1 is the initial version of the node before any editing and may not be undone.

8. `redo()`, which takes an integer parameter indicating a database id number. If `redo_top` is N_A a message is sent to the UI indicating that the "redo" stack is empty. Otherwise, it opens the file indicated by `redo_top`, reads in its content, and sends the content to the UI. Finally, it sets `undo_top` to `redo_top` then increments `redo_top` by one. If `redo_top` is now greater than `num_elements`, `redo_top` is set to N_A.

9. `save()`, which takes a character pointer parameter that points to the new contents of the node. It the string is greater than INT_MAX (defined in limits.h), it returns the code TOO_BIG. Otherwise, it deletes any versions on the "redo" stack and sets `redo_top` to N_A. Then `undo_top` is incremented by one and `num_elements` is set to that number. A new file is then opened using `node_id` and `num_elements` in its name as per the naming convention and the string is written to that file. If the file can not be opened, the function returns CANT_OPEN_TMP_FOR_WRITE, otherwise it returns OK.

10. `new_node_id()`, which takes an integer parameter indicating the offset from the old node id. This member function is called whenever a user commits a modification to the logical view involving a node with a lower id than `node_id`. It resets `node_id` to the old value of `node_id` plus the offset, after moving (renaming) all of the associated /tmp files such that the node id in the naming convention is updated too.

7.2 DB_Queue_Cell and DB_Queue

These classes are used whenever the DB Server must migrate to a new host. From the time a DB Client acknowledges to the old server that it can become the new server until the time it receives some information from the old server, it stores any messages sent to the DB Server on a queue. When the info is received, any messages on the queue are processed one at a time. These classes are put in a separate module from the DB_Server class since the messages are actually enqueued from within module `db_proc_msg.cc`.

The DB_Queue_Cell class has the following private data members :

1. A DCWA_Message pointer called `msg`, which holds the message.

2. A DB_Queue_Cell pointer called `next`, which points to the next cell in the queue.

The DB_Queue_Cell class has the following public member functions :

1. A Constructor which takes a DCWA_Message pointer parameter pointing to the message. It points `msg` to this parameter and sets `next` to NULL.
2. A Destructor which frees the memory associated with `msg`.
3. `get_msg()`, which returns `msg`.
4. `get_next()`, which returns `next`.
5. `set_next()`, which takes a DB_Queue_Cell pointer parameter and sets `next` to that pointer.

The DB_Queue class has the following private data members :

1. A DB_Queue_Cell pointer called `head`, which points to the first cell in the queue.
2. A DB_Queue_Cell pointer called `tail`, which points to the last cell in the queue.

The DB_Queue class has the following public member functions :

1. A Constructor which takes no parameters. It sets `head` and `tail` to NULL.
2. A Destructor which destroys `head` and `tail`.
3. `get_head_msg()`, which returns what the `get_msg()` member function of `head` returns.
4. `enqueue()`, which takes a DCWA_Message pointer that points to the new message to be put on the queue. It instantiates a new DB_Queue_Cell object with the parameter then puts it on the end of the queue.
5. `dequeue()`, which sets `head` to the next cell in the queue then destroys the old cell which `head` pointed to.
6. `empty()`, which returns either TRUE or FALSE depending upon whether or not the queue is empty.

7.3 DB_Query

This class is used to process any queries about the logical view which the user may make. Queries may be "new" or "revised" and have a list of node attributes to display, as well as a criterion which a node must meet in order to be included in the hit list. New queries search the entire logical view but revised queries only search the nodes of the previous query's hit list. Finally, only one criterion is issued per query. If the query is "new", the resulting hit list forms a new hit list. If the query is "revised", the resulting hit list is ANDed to the previous hit list to form the new hit list.

The DB_Query class has the following private data members :

1. A character pointer called `display_list`, which holds a list of node attribute ids, each separated by a space, which the user is interested in and whose values the user wants displayed by the UI.

2. A character pointer called `criteria`, which holds a list of criteria each separated by `&`. If a group member creates a new node (or modifies one) such that it too meets all the previous criteria, it will be included in the hit list as soon as the local DB is informed about that node.

3. A character pointer called `hit_list`, which holds a list of node ids, each separated by a space, that conform to the criteria.

The DB_Query class has the following private member functions :

1. `create_display_list()`, which takes a character pointer parameter that points to a list of node attributes whose values the user wants displayed. This function scans the list and converts each attribute name to an internally maintained attribute id. These attribute ids are stored in `display_list` with the node_id's attribute id always put at the beginning of the list.

2. `update_criteria()`, which takes a character pointer parameter that points to the search criterion of the current query. The function returns TRUE or FALSE depending upon whether the current search criterion is already among the criteria or not. That is, it searches `criteria` and tries to match the current criterion with any previous criteria.

3. `create_hit_list()`, which takes a character pointer parameter that points to the search criterion of a "new" query. It breaks the criterion into three parts, namely an attribute id, an operator id, and a value. It then calls the logical view's root node's `query_node()` member function and stores whatever it returns into `hit_list`.

4. `update_hit_list()`, which takes a character pointer parameter that points to the search criterion of a "revised" query. It first checks if `hit_list` is NULL, in which case it returns since the new criterion cannot limit the hit list even further. Otherwise, it breaks the criterion into three parts, just as `create_hit_list()` did. Then it calls the `query_node()` member function for each node of the previous hit list to determine if it conforms to the new criterion. If it does, it will remain in the hit list, otherwise it will be removed from the hit list.

5. `reconstruct_hit_list()`, which goes through each of the criteria in order to reform the hit list after a group member modifies the logical view. If `criteria` is NULL, `hit_list` is set to NULL, otherwise this function calls `create_hit_list()` for the first criterion and `update_hit_list()` for each subsequent criterion.

The DB_Query class has the following public member functions :

1. A Constructor which takes no parameters. It sets `display_list`, `criteria`, and `hit_list` to NULL.

2. A Constructor which takes two character pointer parameters, pointing to an old display list and old criteria, respectively. This constructor is used when a group member modifies the logical view and the database must reconstruct the user's view based on past queries and the new logical view. It sets `display_list` to point to the first parameter and `criteria` to point to the second parameter, and then calls the `reconstruct_hit_list()` member function.

3. A Destructor which frees the memory associated with `display_list`, `criteria`, and `hit_list`.

4. `get_display_list()`, which returns `display_list`.

5. `get_criteria()`, which returns `criteria`.

6. `create_lists()`, which takes an integer parameter indicating whether the query is a "new" query or a "revised" query and a character pointer that points to the user's

query. It breaks the user's query into two parts, namely the "display" part and the "criteria" part. It then calls the `create_display_list()` member function, passing it the "display" part. If the query is a "new" query, it stores the "criteria" part in `criteria` and then calls the `create_hit_list()` member function, passing it the "criteria" part. If the query is a "revised" query, it calls the `update_criteria()` member function, passing it the "criteria" part. If this function returns FALSE, the `update_hit_list()` member function is called and the "criteria" part is passed to it.

7. `in_hit_list()`, which takes an integer parameter indicating a node id. It returns a code, namely -1 if `hit_list` is NULL, 1 if the node is in the hit list, or 0 if it is not.

8. `form_answer()`, which returns the values of the attributes represented in `display_list` for the nodes in `hit_list`.

7.4 DB_Strtok

This class is used whenever a list (of "words" each separated by a token character not in the "words") needs to be scanned. The standard C function `strtok()` [string token] was not used because :

- It does not function correctly when a member function is called before the next call to `strtok()`.
- It destroys the original string since it stores a NULL character after the first word.
- It allows the "token" to be more than one character and the token may be different from call to call.
- It does not allocate separate memory space for the string that it returns.

The DB_Strtok class has the following private data members :

1. A character pointer called `str`, which holds the string to be scanned.
2. A character pointer called `token`, which holds the separator token.
3. A character pointer called `c1`, which points to the current position in `str` as the scanning proceeds.

The DB_Strtok class has the following public member functions :

1. A Constructor which takes a character pointer parameter pointing to the string to be scanned and a character parameter which indicates the separator token. It creates a new version of the string in memory (pointed to by `str`) then stores the second parameter as a character pointer pointed to by `token`. Thus, internally the token is stored as a character pointer so that it can be used with the standard C string functions but it is passed in as a character so that only one character may be used as a "token".

2. A Destructor which frees the memory associated with `str` and `token`.

3. `get_next_word()`, which returns a character pointer (pointing to a newly allocated memory block) holding the next "word" in the list. When the end of the list is reached, NULL is returned instead.

7.5 LongStringCell and LongString

These classes are used when the user decides to commit his/her edits of a node. The content of the physical file occurring after the node needs to be stored into a temporary buffer so that it is not overwritten. However, since the content may be very large (> 32Kb), a linked list was implemented such that each cell could hold up to 32Kb. Though these classes are only used inside the db_server.cc module, they have been put into a separate file in case in future versions of DCWA all character pointers are converted to LongStrings, so that users are not limited to nodes of size 32K or less.

The LongStringCell class has the following private data members :

1. A character pointer called `data`, which holds the information.

2. A LongStringCell pointer called `next`, which points to the next cell in the list.

The LongStringCell class has the following public member functions :

1. A Constructor which takes an integer parameter indicating the number of characters that `data` will need to hold. It calls `malloc()` to allocate the space for `data` and then sets `next` to NULL.

2. A Destructor which frees the memory associated with `data`.

3. `get_next()`, which returns `next`.

4. `set_next()`, which takes a LongStringCell pointer parameter and sets `next` to that pointer.

5. `read()`, which takes a FILE pointer parameter and an integer parameter. It reads from the current position within the file for as many characters as indicted by the second parameter and puts that information into `data`.

6. `write()`, which takes a FILE pointer parameter. It writes out the information in `data` to the current position within the file.

The LongString class has the following private data member :

1. A LongStringCell pointer called `head`, which points to the first cell in the list.

The LongString class has the following public member functions :

1. A Constructor which takes a long parameter indicating the number of characters that the list will need to hold. If this parameter is greater than LONG_MAX (defined in limits.h), head is set to NULL and the function returns since it is unable to operate with numbers larger than this limit. If the parameter is less than or equal to LONG_MAX, it instantiates `head` and as many other LongStringCells as needed to store the indicted number of characters.

2. A Destructor which destroys each LongStringCell in the list, starting with `head`.

3. `get_head()`, which returns `head`.

4. `read()`, which takes a FILE pointer parameter and two long parameters, indicating the start point and end point, respectively. It calls `fseek()` to set the current position within the file to the indicted start point and then calls the `read()` member function of each LongStringCell in the list until the indicated end point is reached.

5. `write()`, which takes a FILE pointer parameter. It simply calls the `write()` member function of each LongStringCell in the list.

7.6 Update_LV_Cell and Update_LV_List

These classes are used after a group member modifies the logical view such that there are more or less nodes in the tree. The DB Server broadcasts the modification and each DB Client must acknowledge that it received the change. However, before the change was received, a DB Client may have sent a message to the DB Server concerning some node. If the node was part of the modified subtree or after it, the node id in the message will be incorrect. Therefore, these classes are used to correct the node id of the message to the node id which the DB Server is currently storing. Since two or more group members may commit modifications to the logical view before a DB Client acknowledges a modification, a time stamp associated with the change is also part of the acknowledgement and the relevant information about each change is kept in a separate cell of a linked list. When all DB Clients acknowledge the change, the cell is removed.

The Update_LV_Cell class has the following private data members :

1. An array of booleans of size MAX_USERS called `ids`, which stores whether or not a particular DB has acknowledged the modification. TRUE means that the DB Client has not yet acknowledged the modification.

2. An integer called `node_id`, which stores the old id of the first node that is after the subtree that was modified. Messages concerning nodes before this node will not need to have their node id adjusted for processing by the DB Server.

3. An integer called `offset`, which stores the amount by which to adjust a node's id. It may be positive or negative but not zero, since these classes are not needed when the subtree has the same number of nodes after the modification as before it.

4. A `time_t` (long integer) called `commit`, which stores the time stamp associated with the commit of the modification.

5. An Update_LV_Cell pointer called `next`, which points to the next cell in the list.

The Update_LV_Cell class has the following public member functions :

1. A Constructor which takes an array of booleans, two integers, and a `time_t` as parameters. It sets each element of `ids` to the corresponding element of the first parameter. It then sets `node_id` to the second parameter, `offset` to the third parameter, and `commit` to the fourth parameter. Finally, it sets `next` to NULL.

2. `reset_id()`, which takes two integer parameters, representing a DB id and a node id, respectively and a `time_t` parameter, representing a commit time. It sets the element of `ids` associated with the first parameter to FALSE if the second and third parameters are equal to `node_id` and `time_t`, respectively. Finally, it returns TRUE if all the DB Clients have acknowledged this modification or FALSE otherwise.

3. `get_next()`, which returns `next`.

4. `set_next()`, which takes an Update_LV_Cell pointer parameter and sets `next` to that pointer.

5. `update_node_id()`, which takes an integer parameter, representing a DB id and an integer pointer parameter which points to the message's node id. If the DB has not yet acknowledged this modification, the node id will be updated by `offset`.

The Update_LV_List class has the following private data member :

1. An Update_LV_Cell pointer called `head`, which points to the first cell in the list.

The Update_LV_List class has the following public member functions :

1. A Constructor which takes no parameters. It sets `head` to NULL.

2. A Destructor which destroys each Update_LV_Cell in the list, starting with `head`.

3. `add()`, which takes an array of booleans, two integers, and a `time_t` as parameters. It instantiates a new Update_LV_Cell object with the parameters then puts it on the end of the list.

4. `update_node_id()`, which takes an integer parameter, representing a DB id and an integer pointer parameter which points to the message's node id. It calls the `update_node_id()` member function of each cell in the list to get the composite updated node id.

5. `remove()`, which takes two integer parameters, representing a DB id and a node id, respectively and a `time_t` parameter, representing a commit time. It calls the `reset_id()` member function of each cell in the list and removes that cell if the function returned TRUE.

7.7 ValidFile_Cell and ValidFile_List

These classes are used during the verification of a logical view to make sure everything is in the right format. Most of the verifications are straight forward but because the nodes of a physical file may be anywhere within the tree, the verification of juxtaposition requires storing some information until the next node in the same file is found. Each cell holds the name of the file and the last end point that occurred and all the cells are arranged in a linked list.

The ValidFile_Cell class has the following private data members :

1. A character pointer called `name`, which holds the filename.
2. An integer called `end`, which holds the last end point that occurred.
3. A ValidFile_Cell pointer called `next`, which points to the next cell in the list.

The ValidFile_Cell class has the following public member functions :

1. A Constructor which takes a character pointer parameter holding the filename and an integer parameter with an end point. It sets `name` to some new space with the filename in it and sets `end` to the second parameter. Finally, it sets `next` to NULL.
2. A Destructor which frees the memory associated with `name`.
3. `get_name()`, which returns `name`.
4. `get_end()`, which returns `end`.
5. `set_end()`, which takes an integer parameter representing a new end point. It sets `end` to this parameter.
6. `get_next()`, which returns `next`.
7. `set_next()`, which takes a ValidFile_Cell pointer parameter and sets `next` to that parameter.

The ValidFile_List class has the following private data members :

1. A ValidFile_Cell pointer called `first`, which points to the first cell in the list.

2. A ValidFile_Cell pointer called `last`, which points to the last cell in the list.

The ValidFile_List class has the following public member functions :

1. A Constructor which takes no parameters. It sets `first` and `last` to NULL.
2. A Destructor which destroys each ValidFile_Cell in the list, starting with `first`.
3. `add()`, which takes a character pointer parameter holding a filename and an integer parameter with an end point. It instantiates a new ValidFile_Cell with the parameters and puts it on the end of the list.
4. `in_list()`, which takes a character pointer parameter holding a filename. It returns TRUE or FALSE depending upon whether there is a cell in the list for filename or not.
5. `test()`, which takes a character pointer parameter holding a filename and an integer parameter with a start point. It returns TRUE or FALSE depending upon whether or not the second parameter is exactly one more than the `end` data member for the cell with filename. The filename should already have been verified that it is in the list by a call to `in_list()`.
6. `update_endpt()`, which takes a character pointer parameter holding a filename and an integer parameter with the next end point. It resets the `end` data member of the cell with filename to the second parameter. The filename should already have been verified that it is in the list by a call to `in_list()`.

7.8 DB_Node and DB_Root_Node

These classes are used to manipulate the nodes of the logical view. DB_Root_Node derives publicly from DB_Node and is only used because some of its member functions make more sense in that they imply that they operate on the entire tree rather than just one node. For example, `create_tree()` simply calls the root node's `create_node()` member function but it makes more sense to call this function instead of calling the root node's `create_node()` member function directly, which might imply that only the root node was being created. In fact, `create_node()` is recursive in the sense that child nodes are created as necessary and each of their `create_node()` member functions are called in turn. In the following discussion, when a member function is said to be recursive, remember that all objects of the same class share the same code segment (though they have different data areas.) Therefore, the recursive function calls a different object's member function but that function not only has the same name but is stored at the same place in memory.

The DB_Node class has the following protected data members :

1. An integer called `node_id`, which holds the node's id that is maintained by the database.
2. A character pointer called `node_name`, which holds the group's name for the node.
3. A node_level (which is an enumeration type) called `node_lvl`, which holds the node's "level" in the tree – one of ROOT, INTERMEDIATE, or LEAF.
4. A time_t (long integer) called `creation`, which holds when the node was created, as determined from the `time()` system call.
5. A character pointer called `creator`, which holds the user name of the group member which first created the node.
6. A time_t (long integer) called `last_change`, which holds when the node was last committed after being locked for editing, as determined from the `time()` system call.
7. A character pointer called `last_editor`, which holds the user name of the group member who last committed edits of the node.
8. A character pointer called `topic`, which holds a topic for the node, as assigned by the group.
9. A character pointer called `description`, which holds a description for the node, as assigned by the group.
10. A character pointer called `keywords`, which holds a list of keywords for the node, as assigned by the group. Each keyword is separated by a space character.
11. A character pointer called `user_def_attr`, which holds a list of user-defined attributes for the node, as assigned by the group. Each new attribute is separated by a space character.
12. A character pointer called `user_def_attr_values`, which holds a single value, as assigned by the group, for each user-defined attribute above. Each value is separated by a space character and the values are in a one-to-one correspondence with the user-defined attributes, thus creating attribute/value pairs.

13. A character pointer called read_access, which holds a list of usernames, as assigned by the group, indicating who may read the contents of the node. Each username is separated by a space character.

14. A character pointer called edit_access, which holds a list of usernames, as assigned by the group, indicating who may edit the contents of the node. Each username is separated by a space character.

15. A character pointer called lv_mod_access, which holds a list of usernames, as assigned by the group, indicating who may modify the logical view starting with the node as the "root" of the subtree. Each username is separated by a space character.

16. A boolean called local_reader, which indicates whether or not the local user is reading the node.

17. An integer called editor, which holds the database id (possibly N_A) of the current editor or logical view modifier of the node.

18. An integer called lock_mode, which indicates whether the group member specified in editor is editing the contents or modifying the logical view. If editor is N_A, lock_mode will also be set to N_A.

19. A DB_Stack pointer called stack, which points to an object of the DB_Stack class. A stack will be instantiated and used if the local user decides to edit the node or modify the logical view with the node as the "root" of the subtree.

20. A media (which is an enumeration type) called media_type, which holds the node's medium for storing information – one of LOGICAL, TEXT, GRAPHICS, IMAGE, AUDIO, or VIDEO. LOGICAL means the node holds no information in a specific medium and is only used as a logical connector within the tree.

21. A character pointer called filename, which holds the machine name and the filepath to the file where the node's content is physically stored on disk. It will be NULL if the node is a LOGICAL node.

22. An integer called start_point, which holds the start point of the node with respect to the beginning of the physical file. It will be N_A if the node is a LOGICAL node or a currently empty non-logical (text, graphics, etc.) node.

23. An integer called `end_point`, which holds the end point of the node with respect to the beginning of the physical file. It will be N_A if the node is a LOGICAL node or a currently empty non-logical node.

24. A DB_Node pointer called `parent`, which points to a node's parent node or is NULL if the node is the root node of the tree.

25. An integer called `num_children`, which holds the number of child nodes of the node.

26. A DB_Node pointer called `children`, which points to the first child node of the node or is NULL if the node has no children. The children are stored in a block of memory of size DB_Node times `num_children`. Therefore, pointer arithmetic may be used to maneuver through the children.

The DB_Node class has the following protected member functions :

1. `insert()`, which takes an integer parameter indicating the starting node id, an integer pointer parameter indicating the number of child nodes of the node whose `insert()` member function just returned and a character pointer pointer parameter indicating the address of the user-supplied attributes (i.e., the position in the string with the user-supplied attributes as the string is scanned.) It returns a character pointer with the database-supplied attributes inserted among the user-supplied attributes.

2. `check_name()`, which takes an integer parameter indicating a code – one of READ, EDIT, or LV_MOD – and a character pointer parameter holding a username. It determines if the user's name is on the specified access list and returns an integer code – either NOT_FOUND or OK.

The DB_Node class has the following public member functions :

1. A Constructor which takes no parameters. It is used to set dummy values to the node's data members until they are filled in by a call to `create_node()`.

2. A Destructor which frees the memory associated with any of the string attributes and frees the node's stack.

3. Operator `new()`, which takes a size_t parameter indicating the size of memory to allocate. It returns what a call to `malloc()` returns (i.e., a pointer, which must be typecast, to the new memory block.)

4. Operator `delete()`, which takes a void pointer to a block of memory to be freed. It typecasts the block to a character pointer and calls `free()`.

5. `get_node_attrs()`, which packs all of the node's attributes, each separated by `\n`, into a newly allocated block of memory and returns a character pointer pointing to that block.

6. `get_node_name()`, which returns `node_name`.

7. `get_node_level()`, which returns `node_lvl`.

8. `get_creation()`, which returns `creation`.

9. `get_local_reader()`, which returns `local_reader`.

10. `set_local_reader()`, which takes a boolean parameter indicating whether or not the user is reading the node's content. It sets `local_reader` to this parameter.

11. `get_local_reading()`, which returns a character pointer holding the node ids of any nodes that the local user is reading.

12. `get_editor()`, which returns `editor`.

13. `set_editor()`, which takes an integer parameter indicating the new value for `editor`.

14. `get_lock_mode()`, which returns `lock_mode`.

15. `set_lock_mode()`, which takes an integer parameter indicating the new value for `lock_mode`.

16. `get_editors_and_locks()`, which returns a character pointer holding the `node_id`, `editor`, and `lock_mode` for the node and its descendants for each of the nodes currently being edited.

17. `get_num_children()`, which returns `num_children`.

18. `get_stack()`, which returns a pointer to `stack`.

19. `set_stack()`, which takes three integer parameters indicating the `undo_top`, `redo_top`, and `num_elements`, respectively. A stack is then instantiated using the `node_id` and the three parameters.

20. `init_stack()`, which takes a character pointer parameter holding the contents with which to initialize the stack (i.e., store as version one.) It instantiates a new stack and calls its `save()` member function. `init_stack()` returns an integer code indicating one of OUT_OF_MEMORY, TOO_BIG, CANT_OPEN_TMP_FOR_WRITE, or OK.

21. `get_stacks()`, which returns a character pointer parameter holding the `node_id`, `undo_top`, `redo_top`, and `num_elements` for each node currently being edited by the local user.

22. `get_filename()`, which returns `filename`.

23. `get_start_point()`, which returns `start_point`.

24. `get_end_point()`, which returns `end_point`.

25. `get_prior_end_point()`, which takes an integer parameter indicating a node id and a character pointer parameter holding a filename. If its `node_id` is greater than or equal to the first parameter it returns N_A. Otherwise, if it is a leaf node with the same filename, it sets a variable to its `end_point`. Then this function recurses for each of the node's child nodes. It return the greater among its `end_point` or what the child nodes' functions return. Thus, it returns the end point of the most prior non-empty leaf node with the same filename.

26. `set_start_and_end_points()`, which takes two integer parameter indicating a new start and end point, respectively. It sets `start_point` to the first parameter and `end_point` to the second.

27. `get_node_contents()`, which returns a character pointer to a newly allocated block of memory holding the data in the appropriate file from start point to end point inclusively.

28. `create_node()`, which takes a DB_Node pointer parameter indicating the parent of the node, an integer pointer parameter indicating a child node's number of children, and a character pointer parameter holding the contents to scan. It goes through the third parameter picking off fields and setting the appropriate attributes of the node. It sets `parent` to the first parameter. Finally, it returns a character pointer to the contents of the third parameter left unscanned.

29. `send_editor_info()`, which takes two integer parameters indicating the sender database's id and the receiver database's id, respectively. When a DB Client opens a

logical view file which the other members of the group are also using, it must be sent info about the current editors (and logical view modifiers) of each node (if not N_A). This function sends a message to the receiver for every node encountered that someone is editing (or modifying.)

30. `get_lv_info()`, which returns the attributes of all the nodes separated by `%{`, `%%`, and `%}`. It calls `get_node_attrs()` then recurses for each of its children.

31. `get_ui_lv_info()`, which is similar to the above function but returns only the attributes which the UI needs to display the logical view, namely the `node_id`, the `node_name`, and the node's parent's `node_name`.

32. `num_descendants()`, which returns the number of descendants of a node, not just the number of children.

33. `free_strings()`, which frees the memory associated with all the string attributes.

34. `free_stack()`, which destroys `stack`.

35. `delete_descendants()`, which recursively destroys any descendant nodes.

36. `insert_db_attrs()`, which takes an integer parameter indicating the starting node id and a character pointer parameter holding the user-supplied attributes. It returns what a call to `insert()` returns.

37. `merge()`, which takes a character pointer parameter holding the contents of a new subtree of the logical view. It calls `free_strings()` to free any of the node's string attributes then calls `delete_descendants()` to remove the old subtree. Finally, it calls `create_node()` to create the new subtree based on the parameter.

38. `new_node_id()`, which takes an integer parameter indicating the offset from the old node id. It adds the parameter to `node_id`, calls its stack's `new_node_id()` member function, and then recurses for each of its children.

39. `update_borders()`, which takes two integer parameters indicating the node id and the offset, respectively and a character pointer parameter holding a filename. It updates any leaf node's start and end points (unless they are N_A) by offset, if it is in the same file and its node id is greater than the node that was edited, as indicated by the first parameter.

40. find_node(), which takes an integer parameter indicating a node id. It returns a pointer to the node with the corresponding id or NULL if a node with such an id is not found.

41. read_perm(), which takes a character pointer parameter holding a username. If all the node's access lists are empty, it either returns what a call to the parent node's read_perm() member function returns or returns an integer code indicating PERMISSION_DENIED depending upon whether the node has a parent or not. If the read and edit access lists are empty, it returns what a call to the node's lv_mod_perm() member function returns or if only the read access list is empty, it returns what a call to the node's edit_perm() member function returns. These calls indicate the hierarchy of permissions as discussed in the section on the node access policy. However, if the read access list is not empty, a call to check_name() is used to determine if the username is on the list. If so, an integer code indicating OK is returned. If not, this function returns what a call to the node's edit_perm() member function returns.

42. edit_perm(), which takes a character pointer parameter holding a username. If the node's edit and logical view modification access lists are empty, it either returns what a call to the parent node's edit_perm() member function returns or returns an integer code indicating PERMISSION_DENIED depending upon whether the node has a parent or not. If the edit access list is empty, it returns what a call to the node's lv_mod_perm() member function returns. These calls indicate the hierarchy of permissions as discussed in the section on the node access policy. However, if the edit access list is not empty, a call to check_name() is used to determine if the username is on the list. If so, an integer code indicating OK is returned. If not, this function returns what a call to the node's lv_mod_perm() member function returns.

43. lv_mod_perm(), which takes a character pointer parameter holding a username. If the node's logical view modification access list is empty, it either returns what a call to the parent node's lv_mod_perm() member function returns or returns an integer code indicating PERMISSION_DENIED depending upon whether the node has a parent or not. This call indicates the hierarchy of permissions as discussed in the section on the node access policy. However, if the logical view modification access list is not empty, a call to check_name() is used to determine if the username is on the list. If so, an integer code indicating OK is returned. If not, an integer code indicating PERMISSION_DENIED is returned.

44. check_lock(), which takes two integer parameters indicating a code – either EDIT or

LV_MOD – and a database id, respectively. If the first parameter is EDIT, then the function returns either an integer code indicating ALREADY_LOCKED_FOR_EDIT or one indicating OK depending upon whether a user is already editing the node or not. If the first parameter indicates LV_MOD, the function must recurse in order to determine if any of the node's descendants are being edited or are part of a logical view modification. It will return an integer code indicating either ALREADY_LOCKED_FOR_EDIT, ALREADY_LOCKED_FOR_LV_MOD, DESCENDANT_LOCKED_FOR_EDIT, DESCENDANT_LOCKED_FOR_LV_MOD, or OK.

45. `check_unlock()`, which takes an integer parameter indicating a database id. It determines if the user is the one who locked the node and returns an integer code indicating either NOT_EDITOR, NOT_LV_MODIFIER, or OK.

46. `set_locks()`, which takes two integer parameters indicating a code – either EDIT or LV_MOD – and a database id, respectively. This function sets `editor` to the second parameter and `lock_mode` to the first parameter. It then recurses to lock the descendants if the first parameter indicates an LV_MOD lock.

47. `reset_locks()`, which takes an integer parameter indicating a node id. This function is used for resetting specific locks when a user commits (or frees) a node by setting both `editor` and `lock_mode` to N_A. If the `lock_mode` was formerly LV_MOD, it recurses down and up the tree resetting locks as appropriate.

48. `remove_locks()`, which takes an integer parameter indicating a database id. This function is used for removing any locks of a user that quits by setting both `editor` and `lock_mode` to N_A and recursing down the tree.

49. `print_node()`, which takes a FILE pointer parameter indicating the output file. It prints the attributes of the node which need to be saved and recurses down the tree, inserting `%{`, `%%`, and `%}` where appropriate.

50. `get_some_attrs()`, which takes a character pointer parameter holding a list of attribute ids, as described in the section on queries. It appends the values of the specified attributes to a character pointer, which it returns.

51. `int_cmp()`, which takes three integer parameters indicating the value of an attribute to test, an operator id, and the user's test value, respectively. It returns TRUE or FALSE depending upon the outcome of the comparison.

52. `long_cmp()`, which takes a long parameter indicating the value of an attribute to test, an integer parameter indicating an operator id, and a long parameter indicating the user's test value. It returns TRUE or FALSE depending upon the outcome of the comparison.

53. `str_cmp()`, which takes a character pointer parameter holding the value of an attribute to test, an integer parameter indicating an operator id, and a character pointer parameter holding the user's test value. It returns TRUE or FALSE depending upon the outcome of the comparison.

54. `str_eq_cmp()`, which takes the same parameters as `str_cmp()` but the second parameter may only hold a value indicating `==` or `!=`. It returns TRUE or FALSE depending upon the outcome of the comparison.

55. `date_cmp()`, which takes seven integer parameters indicating the value of the month to test, the value of the day to test, the value of the year to test, an operator id, the user's month test value, the user's day test value, and the user's year test value, respectively. It returns TRUE or FALSE depending upon the outcome of the comparison.

56. `list_cmp()`, which takes the same parameters as `str_cmp()` but the first parameter holds a list of words, each separated by a space character, and the second parameter may only hold a value indicating `==` or `!=`. Here `==` indicates "is a member of" and `!=` indicates "is not a member of". This function therefore determines if the user's test value is in the list or not and it returns TRUE or FALSE depending upon the outcome of the comparison.

57. `list_position()`, which takes two character pointer parameters holding a word to look for and a list of words to look in, respectively. It returns the position of the word in the list, with zero indicating the first position, or returns N_A, if the word is not found in the list.

58. `get_list_value()`, which takes a character pointer parameter holding a list of words and an integer parameter indicating a position within the list. It returns the word in the list at the indicated position and assumes that the position is valid, as determined by a previous call to `list_position()`. This function and the previous one are used to manipulate the attribute/value pairs. The user-defined attributes are stored in `user_def_attr` and the associated values are stored in `user_def_attr_values` in a one-to-one correspondence. Thus an attribute's position in `user_def_attr` may be

found by calling `list_position()` and the attribute's corresponding value may be found by calling `get_list_value()`, passing in the position returned from `list_position()`.

59. `query_node()`, which takes two integer parameters indicating an attribute id and an operator id, respectively, a character pointer parameter holding the user's test value, and a boolean parameter indicating whether the function should recurse or not. This function calls one of `int_cmp()`, `long_cmp()`, `str_cmp()`, `str_eq_cmp()`, `date_cmp()`, or `list_cmp()`, depending upon the attribute indicated. It returns a list of node(s) which match the criteria.

The DB_Root_Node class has the following private member function :

1. `validate()` which takes an integer parameter indicating the node id where the validation process should start, a boolean pointer parameter indicating whether a recursive call of this function is returning an error message or the unprocessed part of the logical view, an integer pointer parameter indicating a child node's number of children, and a character pointer holding the logical view to be validated. The first parameter is only used on the first call. The second parameter is set to FALSE when a recursive call of this function is returning the unprocessed part of the logical view and TRUE when it is returning an error message indicating why the logical view is invalid. This function scans the string containing the logical view to make sure it is valid but it does not actually create any of the nodes, therefore this function is truly recursive (i.e., it does not call a different object's member function with the same name.)

   This function checks for the following error conditions :

   - The node's specified id is incorrect with respect to the counter kept by this function.
   - The node's specified name is empty. However, this function does not check if the node's name is only one or more space characters. This may be changed in future versions of the DCWA.
   - The node's specified level is not one of the correct codes – either ROOT, INTERMEDIATE, or LEAF.
   - The node's specified level is incorrect with respect to its position in the tree.
   - The node's specified creation time is less than zero.

- The node's specified creator is empty.
- The node's specified last change time is less than zero.
- The node's specified last editor is empty.
- The node's specified media is not one of the correct codes – either LOGICAL, TEXT, GRAPHICS, IMAGE, AUDIO, or VIDEO.
- The node's specified media is incorrect with respect to its position in the tree. That is, LEAF nodes cannot be LOGICAL nodes and non-logical nodes (i.e., text, graphics, etc.) must be LEAF nodes.
- The node's specified filename is incorrect with respect to its position in the tree. That is, a LOGICAL node should not have an associated filename and a non-logical node should.
- Logical nodes should have their start and end points set to N_A.
- Non-logical nodes may have their start and end points set to N_A if they are empty but the start and end points should never be less than N_A, which is assumed to be -1.
- If a node's specified start point is N_A, it's specified end point should also be N_A, and vice versa.
- If a node's specified start and end points are not N_A, the end point should be greater than or equal to the start point. (If they are equal, it means the node represents only one character.
- Leaf nodes which are not empty and have the same filename should have adjacent start and end points. The first non-empty node with the filename should have a start point of zero and subsequent non-empty nodes with the filename should have a start point that is exactly one more than the end point of the previous non-empty node with the filename. The ValidFile_Cell and ValidFile_List classes are used in conjunction with this test.
- A node's specified number of child nodes cannot be negative.
- The node's specified number of child nodes is incorrect with respect to its position in the tree. That is, leaf nodes cannot have child nodes and non-leaf nodes must have child nodes.
- All the access lists in the tree are empty.
- Unexpected end of logical view.

- Newline character (\n) not found.
- %{, %%, or %} not found.

If one of the above error conditions occurs, a character pointer is returned which points to a message indicating the line number (since fields are separated by \n) of the logical view where the error occurred and a short description of what is wrong. Of course, in some cases the problem may actually be on a line occurring before the indicated line.

Finally, note that the topic, description, keywords, user-defined attributes and user-defined attributes' values fields may be anything, even blank, so no tests are performed on these fields other than making sure the correct number of newline characters are in the logical view.

The DB_Root_Node class has the following public member functions :

1. A Constructor which takes no parameters. It is used to set dummy values to the node's data members until they are filled in by a call to `create_tree()`.

2. A Constructor which takes a character pointer parameter holding a user name. It is used to set dummy values to the node's data members for a default root node. It sets the `creator` and `last_editor` attributes to the character pointer parameter.

3. A Destructor which deletes any of the root node's descendants, frees the memory associated with any of the string attributes, and frees the node's stack.

4. `validate_lv()`, which takes an integer parameter indicating the node id where the validation process should start and a character pointer holding the logical view to be validated. It returns what a call to the root node's `validate()` private member function returns (i.e., a message why the logical view is invalid or an empty string if it is ok.

5. `create_tree()`, which takes a character pointer parameter holding the logical view to be scanned. It calls the root node's `create_node()` member function with NULL as the first parameter, an address of a dummy integer as the second parameter, and its character pointer parameter as the third parameter.

6. `set_editors_and_locks()`, which takes a character pointer parameter holding the `node_id`, `editor`, and `lock_mode` for each node currently being edited. The appropriate data members are set for those nodes.

7. `set_local_reading()`, which takes a character pointer parameter holding the `node_id` for each node that the local user is currently reading. The `local_reader` data member is set for each of those nodes.

8. `set_stacks()`, which takes a character pointer parameter holding the `node_id`, `undo_top`, `redo_top`, and `num_elements` for each node that the local user is currently editing. The `stack` data member is instantiated for each of those nodes.

9. `merge_lvs()`, which takes an integer parameter indicating the node id where the merging process should start and a character pointer holding the contents of the new subtree of the logical view. It calls the `merge()` member function of the specified starting node then resets the node ids of all nodes which occur after the subtree. It returns a pointer to a structure with fields holding the old node id of the node which occurs after the modified subtree (or N_A if there is no such node) and the offset with which node ids after the modified subtree should be corrected.

10. `check_permission()`, which takes an integer parameter indicating which type of permission should be checked (read, edit or logical view modification), a character pointer parameter holding the user's name, and an integer parameter indicating the node id which the user is interested in. If a node with the specified id cannot be found, it will return an integer code indicating NO_SUCH_NODE. Otherwise, it calls the specified node's `read_perm()`, `edit_perm()`, or `lv_mod_perm()` member function and will return the integer code – either PERMISSION_DENIED or OK – which the called function returned.

11. `lock()`, which takes three integer parameters indicating the type of lock (i.e., edit or logical view modification), the database's id, and the node id, respectively. If a node with the specified id cannot be found, it will return an integer code indicating NO_SUCH_NODE. Otherwise, it calls the specified node's `check_lock()` member function. If that function does not return OK, `lock()` returns the integer code which `check_lock()` did return – either ALREADY_LOCKED_FOR_EDIT, ALREADY_LOCKED_FOR_LV_MOD, DESCENDANT_LOCKED_FOR_EDIT, or DESCENDANT_LOCKED_FOR_LV_MOD. Otherwise, it calls the specified node's `set_locks()` member function and returns a code indicating OK.

12. `unlock()`, which takes two integer parameters indicating the database's id and the node id, respectively. If a node with the specified id cannot be found, it will return an integer code indicating NO_SUCH_NODE. Otherwise, it calls the specified node's

check_unlock() member function. If that function does not return OK, unlock() returns the integer code which it did return – either NOT_EDITOR, or NOT_LV_MODIFIER. Otherwise, it calls the specified node's reset_locks() member function and returns a code indicating OK.

13. print_tree(), which takes a FILE pointer parameter indicating the output file. It calls the root node's print_node() member function.

7.9 DB_Client and DB_Server

These classes are used to instantiate a database object on each host. DB_Server derives publicly from DB_Client since the server must act like a client for local messages.

The DB_Client class has the following protected data members :

1. An integer called id, which holds the database's id as assigned by the DB Server.
2. A boolean called soon_to_be_server, which is TRUE if the database has agreed to become the new DB Server during a server migration.
3. A boolean called soon_to_quit, which is TRUE if the database has sent a message to the DB Server informing it that the local user wants to quit.
4. An array of character pointers of size MAX_USERS called usersinfo, which holds the user name and host machine associated with each database currently running in the group. The user name and host machine are separated by a space character.
5. A DB_Root_Node pointer called lv, which points to the root node of the logical view tree.
6. A character pointer called lv_filename, which holds the filename where the logical view is stored.
7. A DB_Query pointer called query, which is used to process any queries that the user may make.

The DB_Client class has the following protected member function :

1. error_recovery(), which looks for DCWA files in the /tmp directory and informs the UI if there are any. The DB cannot do more than this since, even though the node id is

associated with the file, a group member may have changed the logical view since the file(s) were written to disk. Therefore, the user should look at the files and determine what to do with them.

The DB_Client class has the following public member functions :

1. A Constructor which takes no parameters. This constructor is for "normal" DB Clients. It sets the private data members to some default values, then sends a message to the DB Server informing it that there is a new client.

2. A Constructor which takes an integer parameter that is always zero but is used to distinguish this constructor from the previous one. This constructor is for initializing the DB Client data members of the initial DB Server object.

3. A Constructor which takes an integer parameter indicating a database id and a boolean parameter that is always TRUE but is used to distinguish this constructor from the previous one. This constructor is for initializing the DB Client data members of a migrating DB Server.

4. A Destructor which frees the memory associated with `usersinfo` and `lv_filename`, and destroys `lv`.

5. `get_id()`, which returns `id`.

6. `set_id()`, which takes an integer parameter indicating the database id assigned by the DB Server and a character pointer parameter holding information about all the other databases' users and hosts. If the id is MAX_USERS_REACHED, the DB informs the UI and NA about this condition and sets the "quit flag" to TRUE. Otherwise, `id` is set to the first parameter and the information in the second parameter is stored in the appropriate element(s) of `usersinfo`. Finally, it instantiates a default logical view of one node and calls `error_recovery()`

7. `get_soon_to_be_server()`, which returns `soon_to_be_server`.

8. `get_soon_to_quit()`, which returns `soon_to_quit`.

9. `new_peer()`, which takes an integer parameter indicating a database id and a character pointer parameter holding information about a new database's user and host. The information in the second parameter is stored in the appropriate element of `usersinfo`.

10. `new_lv()`, which is a *virtual* function used when the user wants to create a new logical view. It instantiates a default logical view of one node and then initializes the root node's stack with that logical view. Finally, it sends the UI the root node's `node_id`, `node_name`, and an empty field representing the root node's parent's name.

11. `commit_lv_as()`, which is a *virtual* function used when the user wants to commit a newly created logical view under some filename, which is specified in its character pointer parameter. It reads the version of the new logical view that is on top of the root node's "undo" stack and then calls the root node's `insert_db_attrs()` member function to add some other attributes to the nodes beyond what the user supplies. It then calls the root node's `validate_lv()` member function to validate the new logical view. Finally, it writes out the new logical view to the specified file, destroys the logical view, and instantiates a new default logical view, since the user has committed the new one.

12. `commit_lv_leaves()`, which is a *virtual* function used when the user has also broken up preexisting file(s) into certain leaf nodes of the new logical view, whose filename is specified in the character pointer parameter. Since the new logical view was deleted after it was committed, this function rereads it.

13. `commit_lv_leaf()`, which is a *virtual* function that takes an integer parameter indicating a node id and a character pointer parameter holding the contents of the specified node, as broken up from preexisting file(s). The filename associated with the node is opened for appending and the contents are written onto the end of the file. Appending may be used since it is assumed that the UI will send the nodes in a top-down fashion. Finally, the node's start and end points are updated.

14. `commit_lv_last_leaf()`, which is a *virtual* function that takes the same parameters as `commit_lv_leaf()`, which it calls to commit the contents of the last non-empty leaf node. It then writes out the updated logical view to its file and finally sends an acknowledgement to the UI.

15. `request_lv()`, which is a *virtual* function used when the user wants to open the existing logical view that is specified in its character pointer parameter. It sends a message to the DB Server requesting it to open the file since the server must determine if the user has requested the same logical view as the other group members.

16. `receive_lv()`, which takes a character pointer parameter holding the contents of a logical view as sent by the DB Server. It strips off the filename from the parameter

and stores it in `lv_filename` then passes the rest of the contents of the parameter to the root node's `create_tree()` member function. It then calls the root node's `get_ui_lv_info()` member function and passes to the UI what it returns.

17. `err_lv()`, which takes an integer pointer parameter indicating an error code and a character pointer parameter holding the requested filename and the filename of the group's current logical view separated by a space. This function is called when the DB Server sends a message informing the client that it was unable to fulfill the request to open the specified logical view. This function informs the UI about the error.

18. `resend_lv()`, which is a *virtual* function used when the UI needs information about the current logical view, presumably to redraw it. Depending upon the user's previous queries, it will send information to the UI about the entire logical view or just the user's personal view.

19. `request_quit_lv()`, which instantiates a new default logical view and informs the DB Server that the user has closed the group's logical view.

20. `new_query()`, which takes a character pointer parameter holding a query. It instantiates a new DB_Query object then calls its `create_lists()` and `form_answer()` member functions, sending to the UI what the latter function returned.

21. `revised_query()`, which takes a character pointer parameter holding a query. It calls query's `create_lists()` and `form_answer()` member functions, sending to the UI what the latter function returned.

22. `request_editor_info()`, which takes an integer parameter indicating a node id. If the specified node is being edited, it sends the UI the editor's username and host, as well as the mode of editing – EDIT or LV_MOD – in the `err_reason` field. Otherwise, it sends the UI a NO_CURRENT_EDITOR message.

23. `request_node_attrs()`, which takes an integer parameter indicating a node id. It sends the UI all the attributes of the specified node by calling the node's `get_node_attrs()` member function.

24. `request_read()`, which takes an integer parameter indicating a node id. It checks if the user has read permission on the specified node and whether or not it is a leaf node. It then reads the content of the node and sends it to the UI, unless the node is empty, in which case there is nothing for the user to read. If the node is not empty, it calls the

node's `set_local_reader()` member function to store the fact that the local reader is reading the node.

25. `finished_reading()`, which takes an integer parameter indicating a node id. It calls the specified node's `set_local_reader()` member function to store the fact that the local reader is no longer reading the node.

26. `request_edit()`, which is a *virtual* function used when the user wants to edit the node that is specified in its integer parameter. It sends a message to the DB Server requesting it to see if the user can edit the node.

27. `receive_edit()`, which takes an integer parameter indicating a node id and a character pointer parameter holding the content of the node which the DB Server has sent back. It initializes the node's stack with the content and then sends the content to the UI.

28. `err_edit()`, which takes two integer parameters indicating a node id and an error code, respectively. This function is called when the DB Server sends a message informing the client that it was unable to fulfill the request to lock the specified node for editing. This function informs the UI about the error.

29. `new_editor()`, which takes an integer parameter indicating a node id and a character pointer parameter holding the editor's database's id and the mode in which the remote user locked the node. It calls the root node's `unlock()` or `lock()` member function depending upon whether the editor's database's id is N_A or not. It then determines if the local user is reading the node or if the node is in the local user's personal view. In either case, it sends a message to the UI informing it that the node is now being edited by a remote user.

30. `request_lv_mod()`, which is a *virtual* function used when the user wants to modify (part of) the logical view. The subtree will start at the node with the id specified in the integer parameter. The user may also use this locking of the subtree for logical view modification to change node attributes. However, in either case it sends a message to the DB Server requesting it to see if the user can modify the subtree.

31. `receive_lv_mod()`, which takes an integer parameter indicating a node id. This function is called when the DB Server has informed the DB Client that it is ok for the user to modify the logical view subtree starting at that node. This function initializes the node's stack with the current subtree then sends an acknowledgement message to the UI.

32. `err_lv_mod()`, which takes two integer parameters indicating a node id and an error code, respectively. This function is called when the DB Server sends a message informing the client that it was unable to fulfill the request to lock the specified subtree for logical view modification. This function informs the UI about the error.

33. `undo()`, which takes an integer parameter indicating a node id. It calls the specified node's stack's `undo()` member function.

34. `redo()`, which takes an integer parameter indicating a node id. It calls the specified node's stack's `redo()` member function.

35. `save()`, which takes an integer parameter indicating a node id and a character pointer parameter holding the contents of the node to be put onto the stack. It calls the specified node's stack's `save()` member function.

36. `request_commit()`, which is a *virtual* function used when the user wants to commit the edits (or logical view modifications) to the master file. The node id is specified in the integer parameter and this function reads the /tmp file that has the last saved version (i.e., is on top of the "undo" stack.) If the node was locked for logical view modification, the DB adds some other attributes to the nodes beyond what the user supplied and then verifies the logical view for correct format. Finally, it sends a message to the DB Server requesting it to commit the node.

37. `receive_commit()`, which takes an integer parameter indicating a node id. This function destroys the node's stack then informs the UI of the DB Server's acknowledgement of the commit.

38. `err_commit()`, which takes two integer parameters indicating a node id and an error code, respectively. This function is called when the DB Server sends a message informing the client that it was unable to fulfill the request to commit the node. This function informs the UI about the error.

39. `new_node_borders()`, which takes an integer parameter indicating a node id and a character pointer parameter holding the new start and end points and the offset from the old start and end points. It calls the specified node's `set_start_and_end_points()` member function then calls the root node's `update_borders()` member function, which will update any leaf node's start and end points (unless they are N_A) by the offset, if it is in the same file and its id is greater than the node that was edited.

40. `update_lv()`, which takes an integer parameter indicating a node id and a character pointer parameter holding the commit time and contents of a new subtree of the logical view. It strips off the commit time and sends an acknowledgement message to the DB Server then merges the new subtree with the existing logical view. Depending upon the user's previous queries, it then either sends the UI what it needs in order to display the entire new logical view or reconstructs the user's personal view and sends the UI that information.

41. `quit_node()`, which is a *virtual* function used when the user wants to quit the node specified in the integer parameter without committing the changes. It destroys the node's stack and then informs the DB Server about the user's decision.

42. `send_local_info()`, which returns a character pointer holding the ids for the nodes the local user is reading, the stack info for the nodes the local user is editing, and any query information.

43. `request_quit()`, which sets `soon_to_quit` to TRUE and sends a message to the DB Server informing it about the user's decision, unless this DB Client has already agreed to become the new DB_Server, in which case it will quit after it receives the information from the old DB Server, processes any accumulated messages on the queue, and finds a new DB Client to migrate to.

44. `quit()`, which is called when the DB Server acknowledges the clients quit request. It sends a message to the NA telling it to quit and then sets the "quit flag" to TRUE.

45. `peer_quit()`, which takes an integer parameter indicating a database id and is called whenever the DB Server informs this client that one of its peer databases has quit. It frees the memory associated with the specified element of `usersinfo` then calls the logical view's root node's `remove_locks()` member function.

The DB_Client class has the following friend functions :

1. `get_users_info()`, which takes an integer parameter indicating a database id. It returns the specified element of `usersinfo`.

2. `get_logical_view()`, which returns `lv`.

The DB_Server class has the following private data members :

1. A boolean array of size MAX_USERS called `ids`, which stores whether or not a particular id has been assigned during the working session, as described in the section on the id policy.

2. A boolean array of size MAX_USERS called `sids`, which maintains the ids of the DB Clients which may potentially become the new DB Server during a server migration.

3. A boolean array of size MAX_USERS called `group_lv`, which maintains the ids of the databases whose user has opened the group's logical view file.

4. A DB_Root_Node pointer called `new_lv_tree`, which is used when the local user whose database is the DB Server wants to create a new logical view. This pointer is used rather than `lv` since `lv` must be used to maintain the other group member's edit locks, etc.

5. A character pointer called `new_lv_filename`, which is used to store the filename of a new logical view until some of its leaf nodes' contents may be committed also, since the user also broke up preexisting file(s) at the same time as creating the new logical view.

6. A boolean called `making_new_lv`, which is TRUE if the local user is creating a new logical view.

7. A DB_Queue pointer called `queue`, which is used to hold any messages that a new DB Server receives before it gets some information from the old DB Server during a server migration.

8. An Update_LV_List pointer called `update_ack`, which is used to modify a message's node id field until a DB Client acknowledges a logical view modification.

The DB_Server class has the following private member functions :

1. `process_edit_commit()`, which takes two integer parameters indicating a database id and a node id, respectively and a character pointer parameter holding the contents of the node to be committed. If the node is currently empty, this function must find out the end point of the most prior non-empty node with the same filename, as returned by the root node's `get_prior_end_point()` member function. This function then reads

in the content of the file which occurs after the indicated node by using an object of the LongString class. It then prints the contents held in the third parameter at the correct place within the file and replaces what was temporarily stored in the LongString object. It then recalculates the end point (and start point if it was N_A) for the node and stores the resulting offset from the previous end point. It then updates any leaf node's start and end points (unless they are N_A) by the offset, if the leaf node is in the same file and its id is greater than the node that was just committed (i.e., it occurs after the current node.) Finally, it informs all the DB Clients of the new start and end points of the node, as well as the offset, unlocks the node, and informs all the DB Clients of the unlock.

2. `process_lv_commit()`, which takes two integer parameters indicating a database id and a node id, respectively and a character pointer holding the contents of the logical view to be committed. This function opens the file where the logical view is stored, merges the new subtree with the existing logical view, and then writes out the result by calling the root node's `print_tree()` member function. The contents of the new subtree are assumed to be valid since they were validated in either the DB Server's or the DB Client's `request_commit()` member function. It then sends the UI information about the entire new logical view or the local user's personal view as adjusted under the new subtree. It then adjusts `update_ack` to keep track of clients that have not acknowledged the update. Finally, it informs all the DB Clients of the new logical view, unlocks the node, and informs all the DB Clients of the unlock.

The DB_Server class has the following public member functions :

1. A Constructor which takes no parameters. This constructor is for the initial DB Server of the group's working session. It sets all elements of `ids` to FALSE except for element zero, which represents the server itself. It sets the other private data members to some default values and then calls `error_recovery()`.

2. A Constructor which takes an integer parameter indicating a database id. This constructor is for initializing some of the DB Server data members when the DB Server migrates to another host. In particular, it instantiates a DB_Queue object for storing any messages it receives until it gets the information from the old server that it needs in order to function as the server. The other DB_Server data members will be initialized when this new DB Server receives the information.

3. A Destructor which destroys `new_lv_tree`.

4. get_making_new_lv(), which returns making_new_lv.
5. get_queue(), which returns queue.
6. new_db_client(), which takes a character pointer parameter holding information about the new user. It determines if all the database ids have been assigned or not. If they have, it sends a message to the new client with a code indicating MAX_USERS_REACHED. Otherwise, it sends a message to the new client informing it of its id, as well as telling it about any other existing databases operating in the group. Finally, it stores the parameter into the appropriate element of usersinfo then informs all the other databases in the group about the new client.
7. open_lv(), which takes an integer parameter indicating a database id and a character pointer parameter holding a filename. If a logical view for the group has not been opened, it opens the file, reads it in, instantiates a new DB_Root_Node for lv, and validates the contents of the file to determine if it really does hold a logical view or not. If it does, lv's create_tree() member function is called to establish the structure. Finally, it returns the logical view to the requesting database. If a logical view for the group has already been opened and the filenames match, it will send the logical view to the requesting database, as well as the current editor and lock_mode information. If a group logical view file has been opened but the new request is to open another file, it informs the requesting database that the file chosen is not the one the group is currently using as the logical view.
8. quit_lv(), which takes an integer parameter indicating a database id. It resets the appropriate element of group_lv and will instantiate a default logical view of one node if all the group members have closed the file.
9. process_edit_request(), which takes two integer parameters indicating a database id and a node id, respectively. It gets the username from usersinfo for the id in the first parameter then determines if the user has edit permission on the node or not. If he/she does, then it will determine if the node is unlocked or not and whether it is a leaf node or not. If these tests are passed, it will lock the node, send the content of the node to the requesting database, and inform all the DB Clients of the new editor.
10. process_lv_mod_request(), which takes two integer parameters indicating a database id and a node id, respectively. It gets the username from usersinfo for the id in the first parameter then determines if the user has logical view modification permission on

the node or not and whether the node is unlocked or not. If these tests are passed, it will lock the node, send an acknowledgement to the requesting database, and inform all the DB Clients of the new logical view modifier.

11. `free_node()`, which takes two integer parameters indicating a database id and a node id, respectively. It unlocks the specified node then broadcasts the unlock to all the DB Clients.

12. `quit_new_lv()`, which is called when the local user wants to quit a logical view creation without committing the changes. It destroys `new_lv_tree` and sets `making_new_lv` to FALSE.

13. `undo_new_lv()`, which calls the `new_lv_tree's` root node's stack's `undo()` member function.

14. `redo_new_lv()`, which calls the `new_lv_tree's` root node's stack's `redo()` member function.

15. `save_new_lv()`, which takes a character pointer parameter holding the contents of the new logical view to be saved. It calls the `new_lv_tree's` root node's stack's `save()` member function.

16. `update_node_id()`, which takes an integer parameter indicating a database id and an integer pointer parameter holding the node id to update. It calls `update_ack's` `update_node_id()` member function.

17. `process_commit()`, which takes two integer parameters indicating a database id and a node id, respectively and a character pointer parameter holding the content of the node or logical view to be committed. It checks the lock on the node against the first parameter then calls either `process_edit_commit()` or `process_lv_commit()`.

18. `update_lv_ack()`, which takes two integer parameters indicating a database id and a node id, respectively and a character pointer parameter holding a commit time. It calls `update_ack's` `remove()` member function.

19. `client_quitting()`, which takes an integer parameter indicating a database id. It sets the appropriate element of `ids` to FALSE, frees the memory associated with the appropriate element of `usersinfo`, removes any locks which the user may still have, calls `update_ack's` `remove()` member function, and sends an acknowledgement to the client. Finally, it tells the other databases about the quitter.

20. `migrate()`, which takes an integer parameter holding either N_A or a database id. It returns TRUE or FALSE depending upon whether there is a DB Client that the server can migrate to or not.

21. `send_server_info()`, which takes an integer parameter indicating a database id. It sends all relevant information to the new DB Server that it will need in order to function as the server.

22. `recv_server_info()`, which takes a character pointer holding the information from the the old DB Server sent in `send_server_info()`. It breaks the information up into pieces and sets the appropriate DB_Server data members accordingly.

23. `recv_local_info()`, which takes a character pointer holding the information concerning what the local user was doing before this database became the DB Server.

24. `process_queue()`, which processes any messages that came to the new DB Server while it was waiting for the old DB Server's information. It processes each message on the queue and removes it.

The DB_Server class redefines the following (public) DB_Client virtual member functions :

1. `new_lv()`, which is used to set up `new_lv_tree` so that the local user may create a new logical view. This function is used so that the group may still collaborate using the logical view that is in data member `lv`.

2. `commit_lv_as()`, which takes a character pointer parameter holding a filename. This function is used to store the local user's newly created logical view (which is on top of the `new_lv_tree's` root node's "undo" stack) under some filename.

3. `commit_lv_leaves()`, which is used when the local user has also broken up preexisting file(s) into certain leaf nodes of the new logical view, whose filename is specified in the character pointer parameter. Since the new logical view was deleted after it was committed, this function rereads it then stores the filename in a new memory block pointed to by `new_lv_filename`.

4. `commit_lv_leaf()`, which takes an integer parameter indicating a node id and a character pointer parameter holding the contents of the specified node, as broken up from preexisting file(s). The filename associated with the node is opened for appending and

the contents are written onto the end of the file. Appending may be used since it is assumed that the UI will send the nodes in a top-down fashion. Finally, the node's start and end points are updated.

5. `commit_lv_last_leaf()`, which takes the same parameters as `commit_lv_leaf()`, which it calls to commit the contents of the last non-empty leaf node. It then writes out the updated logical view to its file and finally sends an acknowledgement to the local UI.

6. `request_lv()`, which takes a character pointer parameter holding a filename. This function calls `open_lv()` directly without having to send a message as a DB Client would.

7. `resend_lv()`, which sends the UI information about `new_lv_tree` if the local user is creating a new logical view or calls the DB_Client's version of `resend_lv()` otherwise.

8. `request_edit()`, which takes an integer parameter indicating a node id. This function calls `process_edit_request()` directly without having to send a message as a DB Client would.

9. `request_lv_mod()`, which takes an integer parameter indicating a node id. This function calls `process_lv_mod_request()` directly without having to send a message as a DB Client would.

10. `request_commit()`, which takes an integer parameter indicating a node id. This function gets the last saved version off the node's stack, then calls `process_commit()` directly without having to send a message as a DB Client would.

11. `quit_node()`, which takes an integer parameter indicating a node id. This function is called when the local user wants to quit the specified node without committing the changes. It destroys the node's stack but does not send a message as a DB Client would.

The next section discusses the modules of the second version's database.

8 Modules

Most of the modules of the second version's database correspond to the classes just discussed. The classes in each of the previous subsections are found in a separate module with two exceptions :

- The ValidFile_Cell and ValidFile_List classes are found in the same module as the DB_Node and DB_Root_Node classes since they are only used by the `validate()` member function of the DB_Root_Node class.
- The DB_Client and DB_Server classes are found in separate modules.

These "class" modules will not be discussed further but there are four other modules of the database :

1. `db.c` contains a function called `db_process()`, which is the function called after the DB is forked off by the CR process. It closes unused pipe ends and gets the various pids just as the prototype's module of the same name. It calls `db_main()` and closes any open pipe ends when that function returns.

2. `db_main.cc` contains a function called `db_main()`, which calls `db_init()` then goes into an infinite loop. Inside this loop the UNIX `select` system call is used to determine if there are any messages in any of the pipes just as the prototype's module of the same name. When a message is received, it is read and passed to `db_process_message()`. When that function returns, the memory associated with the message is freed and if a "quit flag" is set, the loop is broken out of and `db_main()` exits. Note that `db_main()` is the only database function which may be called externally (by `db_process()`, for example) and has been declared as `extern "C"`.

3. `db_proc_msg.cc` contains the functions which process the event-driven messages, as well as two friend functions of the DB_Client class used during queries, as discussed above. Unlike the above two modules, this module differs significantly from the prototype's module of the same name. The main difference is that instead of one large "switch" statement, the process must decide if it is a DB Client or the DB Server before processing the message.

   Different functions are called depending upon the various "states" which the DB may be in :

   (a) The DB process is a new DB Server waiting for information from the old DB Server and is meanwhile putting any other messages on a queue for later processing.

   (b) The DB process is the DB Server but the message is from a local sibling process and so it must act like a DB Client.

(c) The DB process is the DB Server and the message is from a remote process and so it must act as the DB Server.

(d) The DB process is a DB Client but it is still waiting for an id from the DB Server and may not respond to messages until the id is received.

(e) The DB process is a normal DB Client.

The functionality of each of these five cases is put into different functions inside this module and they will not be discussed further except to say that each only responds to certain messages and ignores any others.

However, before any of the above functions will work, two global variables (a DB_Client pointer called `dbc` and a DB_Server pointer called `dbs`) must be initialized. The `db_init()` function will instantiate a new DB Server if the "firsthost" flag is set or will instantiate a new DB Client if it is not set. The appropriate pointer is set to the new object and the other one is set to NULL.

4. `db_send.cc` contains a function called `db_send_to_sibling()`, which uses its parameters to send a message to a sibling (CR, UI, or NA) of the database. The message structures used are as follows :

```
typedef struct _pipe_message
{
  int sender_process;
  int sender_id;
  int receiver_process;
  int receiver_id;
  int code;
  int node_id;
  int err_reason;        /* Error code. */
  int datalength;
} Pipe_Message;

typedef struct _dcwa_message
{
  Pipe_Message aux;
  char *data;
} DCWA_Message;
```

The `data` field is separated from the other fields so that the process on the read end of a pipe may know how much memory to request (as specified in the `datalength` field.) That is, a Pipe_Message pointer is typecast to a character pointer and written into the pipe then the `data` field is written into the pipe after it. On the read end of the pipe, the information from the first write is typecast back into a Pipe_Message pointer and the `datalength` field is used to allocate memory for `data`, which is then read.

Finally, only one difficult integration problem occurred when these DB modules were linked together with the modules of the other DCWA processes, most of which were written in `C`. Although [39] discusses the use of `extern "C"` when linking together modules written in `C` and `C++`, neither there nor anywhere else has it been mentioned that the `main()` function must be compiled with a `C++` compiler. Once this fact was discovered by the author, the integration of the second version of DCWA was completed without difficulty.

The next section gives several examples using the second version's database.

9 Examples

This section gives detailed examples of some of the database's functionality, namely checking users' node access permissions, insertion of the DB maintained attributes within those attributes obtained from the user, queries and personal views, and a semantic network. The logical view tree used in the first three examples is as follows :

```
0 ____
      |
      | - 1 ____
      |          | - 2
      |          |
      |          | - 3
      |
      | - 4 ____
                 | - 5
                 |
                 | - 6
                 |
                 | - 7
```

In the examples that follow, only those node attributes which are relevant to the example will be shown, not all the attributes. Even so, some of the examples need to show many node attributes and the reader may loose sight of a node. Remember that each node (except the root node) has at least one of `%{`, `%%`, and `%}` on either side of it.

9.1 Checking Users' Node Access Permissions

Assume that the logical view file contains the following information for the node id and the read, edit and logical view modification access lists :

```
0
JamesCross StephenSeidman
ShefaliGajiwala SatishAmbati TimDollar ByongLee WesleyWear
YuGong KaiChang
%{
1
JamesCross StephenSeidman SatishAmbati TimDollar ByongLee WesleyWear

YuGong KaiChang ShefaliGajiwala
%{
2




%%
3




%}
4
JamesCross StephenSeidman ShefaliGajiwala SatishAmbati TimDollar ByongLee

YuGong KaiChang WesleyWear
%{
5
```

```
%%
6
```

```
%%
7
```

```
%}
%}
```

The following requests for read, edit, or logical view modification would then result in the associated answers :

1. WesleyWear wants to read node 12. [Though it will probably be impossible for the user to select an invalid node id from within the UI, the DB checks for this condition anyway.]

   Answer :

   ```
   No such node.
   ```

2. JohnDoe wants to read node 0. [Currently the daemon allows anyone to join a group without any security checks, so the existence of the access list attributes supply this need.]

   Answer :

   ```
   Permission denied.
   ```

3. ShefaliGajiwala wants to read node 0. [Only leaf nodes contain readable and/or editable material.]

   Answer :

```
Not a leaf node.
```

4. ShefaliGajiwala wants to read node 2. [A user may read any leaf node for which he/she has permission.]

   Answer :

   ```
   Ok.
   ```

5. ShefaliGajiwala wants to edit node 3. [The same user can have open as many different nodes as needed and in either mode : reading or editing.]

   Answer :

   ```
   ShefaliGajiwala has locked node 3 for editing.
   ```

6. WesleyWear wants to edit node 5.

   Answer :

   ```
   WesleyWear has locked node 5 for editing.
   ShefaliGajiwala has locked node 3 for editing.
   ```

7. YuGong wants to edit node 5.

   Answer :

   ```
   YuGong can not lock node 5 because it is already locked
   for editing.
   WesleyWear has locked node 5 for editing.
   ShefaliGajiwala has locked node 3 for editing.
   ```

8. YuGong wants to edit node 6.

   Answer :

   ```
   WesleyWear has locked node 5 for editing.
   ShefaliGajiwala has locked node 3 for editing.
   YuGong has locked node 6 for editing.
   ```

9. JohnDoe wants to read node 7.

   Answer :

```
Permission denied.
WesleyWear has locked node 5 for editing.
ShefaliGajiwala has locked node 3 for editing.
YuGong has locked node 6 for editing.
```

10. KaiChang wants to read node 7.

    Answer :

```
Ok.
WesleyWear has locked node 5 for editing.
ShefaliGajiwala has locked node 3 for editing.
YuGong has locked node 6 for editing.
```

11. JamesCross wants to read node 7. [More than one user may read the same node at the same time.]

    Answer :

```
Ok.
WesleyWear has locked node 5 for editing.
ShefaliGajiwala has locked node 3 for editing.
YuGong has locked node 6 for editing.
```

12. ShefaliGajiwala wants to edit node 2. [A user may decide to edit a node which he/she is reading, as long as he/she has permission and no other user has locked the node for editing or logical view modification.]

    Answer :

```
WesleyWear has locked node 5 for editing.
ShefaliGajiwala has locked node 2 for editing.
ShefaliGajiwala has locked node 3 for editing.
YuGong has locked node 6 for editing.
```

13. WesleyWear wants to edit node 7. [A user may choose to edit a node even if other users are already reading it. Each of those users will then need to decide if he/she wants to continue reading the last committed version or watch the new edits as they happen.]

    Answer :

    ```
    WesleyWear has locked node 5 for editing.
    WesleyWear has locked node 7 for editing.
    ShefaliGajiwala has locked node 2 for editing.
    ShefaliGajiwala has locked node 3 for editing.
    YuGong has locked node 6 for editing.
    ```

14. WesleyWear wants to read node 2. [A user may decide to read a node that is currently being edited. That user should decide if he/she wants to read the last committed version or watch the new edits as they happen but may toggle to the other option at any time.]

    Answer :

    ```
    Ok.
    WesleyWear has locked node 5 for editing.
    WesleyWear has locked node 7 for editing.
    ShefaliGajiwala has locked node 2 for editing.
    ShefaliGajiwala has locked node 3 for editing.
    YuGong has locked node 6 for editing.
    ```

15. KaiChang wants to modify the logical view starting at node 0.

    Answer :

    ```
    KaiChang can not lock node 0 because a descendant is locked
    for editing.
    WesleyWear has locked node 5 for editing.
    WesleyWear has locked node 7 for editing.
    ShefaliGajiwala has locked node 2 for editing.
    ShefaliGajiwala has locked node 3 for editing.
    YuGong has locked node 6 for editing.
    ```

16. KaiChang wants to modify the logical view starting at node 4.

    Answer :

    ```
    KaiChang can not lock node 4 because a descendant is locked
    for editing.
    WesleyWear has locked node 5 for editing.
    WesleyWear has locked node 7 for editing.
    ShefaliGajiwala has locked node 2 for editing.
    ShefaliGajiwala has locked node 3 for editing.
    YuGong has locked node 6 for editing.
    ```

17. KaiChang wants to modify the logical view starting at node 7.

    Answer :

    ```
    KaiChang can not lock node 7 because it is already locked
    for editing.
    WesleyWear has locked node 5 for editing.
    WesleyWear has locked node 7 for editing.
    ShefaliGajiwala has locked node 2 for editing.
    ShefaliGajiwala has locked node 3 for editing.
    YuGong has locked node 6 for editing.
    ```

18. ShefaliGajiwala releases node 3.

    Answer :

    ```
    WesleyWear has locked node 5 for editing.
    WesleyWear has locked node 7 for editing.
    ShefaliGajiwala has locked node 2 for editing.
    YuGong has locked node 6 for editing.
    ```

19. KaiChang wants to modify the logical view starting at node 3.

    Answer :

```
WesleyWear has locked node 5 for editing.
WesleyWear has locked node 7 for editing.
ShefaliGajiwala has locked node 2 for editing.
YuGong has locked node 6 for editing.
KaiChang has locked node 3 for logical view modification.
```

20. WesleyWear releases node 5.

    Answer :

```
WesleyWear has locked node 7 for editing.
ShefaliGajiwala has locked node 2 for editing.
YuGong has locked node 6 for editing.
KaiChang has locked node 3 for logical view modification.
```

21. ShefaliGajiwala releases node 2.

    Answer :

```
WesleyWear has locked node 7 for editing.
YuGong has locked node 6 for editing.
KaiChang has locked node 3 for logical view modification.
```

22. WesleyWear wants to modify the logical view starting at node 0.

    Answer :

```
Permission denied.
WesleyWear has locked node 7 for editing.
YuGong has locked node 6 for editing.
KaiChang has locked node 3 for logical view modification.
```

23. YuGong wants to modify the logical view starting at node 0.

    Answer :

```
YuGong can not lock node 0 because a descendant is locked
for logical view modification.
WesleyWear has locked node 7 for editing.
YuGong has locked node 6 for editing.
KaiChang has locked node 3 for logical view modification.
```

24. WesleyWear releases node 7.

Answer :

```
YuGong has locked node 6 for editing.
KaiChang has locked node 3 for logical view modification.
```

25. KaiChang wants to modify the logical view starting at node 1. [A logical view modification lock also locks all descendants of the node.]

Answer :

```
YuGong has locked node 6 for editing.
KaiChang has locked node 1 for logical view modification.
KaiChang has locked node 2 for logical view modification.
KaiChang has locked node 3 for logical view modification.
```

26. YuGong releases node 6.

Answer :

```
KaiChang has locked node 1 for logical view modification.
KaiChang has locked node 2 for logical view modification.
KaiChang has locked node 3 for logical view modification.
```

27. KaiChang wants to modify the logical view starting at node 0.

Answer :

```
KaiChang has locked node 0 for logical view modification.
KaiChang has locked node 1 for logical view modification.
KaiChang has locked node 2 for logical view modification.
KaiChang has locked node 3 for logical view modification.
KaiChang has locked node 4 for logical view modification.
KaiChang has locked node 5 for logical view modification.
KaiChang has locked node 6 for logical view modification.
KaiChang has locked node 7 for logical view modification.
```

28. KaiChang releases node 1. [A logical view modification unlock not only unlocks descendant nodes (in this case 2 and 3) but also unlocks any ancestor nodes (in this case 0) so as to maintain only a lock on the minimum number of nodes necessary.]

    Answer :

    ```
    KaiChang has locked node 4 for logical view modification.
    KaiChang has locked node 5 for logical view modification.
    KaiChang has locked node 6 for logical view modification.
    KaiChang has locked node 7 for logical view modification.
    ```

29. KaiChang releases node 7.

    Answer :

    ```
    KaiChang has locked node 5 for logical view modification.
    KaiChang has locked node 6 for logical view modification.
    ```

30. KaiChang releases node 5.

    Answer :

    ```
    KaiChang has locked node 6 for logical view modification.
    ```

31. KaiChang releases node 6.

    Answer :

    ```
    Ok.
    ```

Finally, though not shown in this example, the DB should also be informed by the UI when a user finishes reading a node (closes the window).

9.2 Insertion of the Database-Maintained Attributes

This example is intended to show what would happen if a user were to lock the subtree starting at node 4 for a logical view modification. The user would then make changes, save, undo, and redo until the subtree is as he/she wants it and would finally commit the changes. First, the subtree is shown as it is after the user has finished with it but before the

DB has inserted some attribute's values into the stream of information which the UI sends (representing the attribute's values as filled in by the user.) Then the subtree is shown as it is after the DB has inserted the values for the attributes that it maintains. See the section on node attributes for the list of those attributes which the group is responsible for and those which the DB itself maintains. The overall order of the attributes within the stream is also given in that section. This example also shows that the user has added one more child node to node 4, which was the purpose of his/her logical view modification.

After the user has finished but before the DB has inserted :

```
DB file.
dcwa database
This is a text description for the DCWA DB.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
0

3
%{
DB file part1.
dcwa database
This is a text description for the DCWA DB's first part.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
1
db.c
0
%%
DB file part2.
```

```
dcwa database
This is a text description for the DCWA DB's second part.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
1
db.c
0
%%
DB file part3.
dcwa database
This is a text description for the DCWA DB's third part.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
1
db.c
0
%%
DB file part4.
dcwa database
This is a text description for the DCWA DB's fourth part.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
1
db.c
```

```
0
%}
```

After the DB has inserted what it needs to :

```
4
DB file.
1
782940337
WesleyWear
782940337
WesleyWear
dcwa database
This is a text description for the DCWA DB.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
0

-1 -1
3
%{
5
DB file part1.
2
782940337
WesleyWear
782940337
WesleyWear
dcwa database
This is a text description for the DCWA DB's first part.
CSCW Multimedia Database
```

```
ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
1
db.c
-1 -1
0
%%
6
DB file part2.
2
782940337
WesleyWear
782940337
WesleyWear
dcwa database
This is a text description for the DCWA DB's second part.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
1
db.c
-1 -1
0
%%
7
DB file part3.
2
782940337
WesleyWear
782940337
WesleyWear
```

```
dcwa database
This is a text description for the DCWA DB's third part.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
1
db.c
-1 -1
0
%%
8
DB file part4.
2
782940337
WesleyWear
782940337
WesleyWear
dcwa database
This is a text description for the DCWA DB's fourth part.
CSCW Multimedia Database


ShefaliGajiwala SatishAmbati TimDollar ByongLee JamesCross StephenSeidman

YuGong KaiChang WesleyWear
1
db.c
-1 -1
0
%}
```

9.3 Queries/Personal Views

Assume that the logical view file contains the following information for the node id, node name, creation time, creator, last edit time, last editor, topic, description, keywords, and user-defined attribute/value pairs (with the creation and last edit times also shown "expanded" inside parentheses for the benefit of the reader, though they are stored internally only as long integers) :

```
0
Our files.
772731374 (Mon Jun 27 15:36:14 1994)
KaiChang
772731374 (Mon Jun 27 15:36:14 1994)
KaiChang
dcwa
        This is a text description for the DCWA.
CSCW Multimedia GUI database
new_attribute1 new_attribute2
new_value1 new_value2
%{
1
GUI file.
772731374 (Mon Jun 27 15:36:14 1994)
KaiChang
772899765 (Wed Jun 29 14:22:45 1994)
ShefaliGajiwala
dcwa gui
        This is a text description for the DCWA GUI.
CSCW Multimedia GUI


%{
2
GUI file part1.
772899765 (Wed Jun 29 14:22:45 1994)
ShefaliGajiwala
```

```
773705734 (Fri Jul  8 22:15:34 1994)
ShefaliGajiwala
dcwa gui
        This is a text description for the DCWA GUI's first part.
CSCW Multimedia GUI


%%
3
GUI file part 2.
772899765 (Wed Jun 29 14:22:45 1994)
ShefaliGajiwala
777994140 (Sat Aug 27 13:29:00 1994)
ShefaliGajiwala
dcwa gui
        This is a text description for the DCWA GUI's second part.
CSCW Multimedia GUI


%}
4
DB file.
772731374 (Mon Jun 27 15:36:14 1994)
KaiChang
772799765 (Tue Jun 28 10:36:05 1994)
WesleyWear
dcwa database
        This is a text description for the DCWA DB.
CSCW Multimedia Database


%{
5
DB file part1.
772799765 (Tue Jun 28 10:36:05 1994)
WesleyWear
```

```
778865743 (Tue Sep  6 15:35:43 1994)
WesleyWear
dcwa database
        This is a text description for the DCWA DB's first part.
CSCW Multimedia Database


%%
6
DB file part2.
772799765 (Tue Jun 28 10:36:05 1994)
WesleyWear
776793761 (Sat Aug 13 16:02:41 1994)
WesleyWear
dcwa database
        This is a text description for the DCWA DB's second part.
CSCW Multimedia Database


%%
7
DB file part3.
772799765 (Tue Jun 28 10:36:05 1994)
WesleyWear
781347902 (Wed Oct  5 09:05:02 1994)
WesleyWear
dcwa database
        This is a text description for the DCWA DB's third part.
CSCW Multimedia Database


%}
%}
```

The following queries would then result in the associated answers :

1. This New Query results in the specified attributes (headed by the node id) to be returned to the UI for nodes 4 through 7 (each separated by %%.) Note that the user-defined attribute/value pairs are empty for these four nodes.

   ```
   DISPLAY : Node Name,Description,User Attribute-Value Pairs
   FOR : Node Id > 3
   ```

   Answer :

   ```
   4
   DB file.
           This is a text description for the DCWA DB.


   %%
   5
   DB file part1.
           This is a text description for the DCWA DB's first part.


   %%
   6
   DB file part2.
           This is a text description for the DCWA DB's second part.


   %%
   7
   DB file part3.
           This is a text description for the DCWA DB's third part.
   ```

2. This Revised Query changes some of the attributes the user wishes to see and also further limits the personal view to only those nodes (from the previous answer) whose ids are less than 7 :

```
DISPLAY : Node Name,Creator,Topic,Description
FOR : Node Id < 7
```

Answer :

```
4
DB file.
KaiChang
dcwa database
        This is a text description for the DCWA DB.
%%
5
DB file part1.
WesleyWear
dcwa database
        This is a text description for the DCWA DB's first part.
%%
6
DB file part2.
WesleyWear
dcwa database
        This is a text description for the DCWA DB's second part.
```

3. This is a New Query so the entire tree is searched to created the answer, not just the previous hit list. Note that the attribute used in the criteria (Keywords) need not also be listed in the "display part" :

```
DISPLAY : Node Name,Description
FOR : Keywords == GUI
```

Answer :

```
0
Our files.
        This is a text description for the DCWA.
%%
```

```
1
GUI file.
        This is a text description for the DCWA GUI.
%%
2
GUI file part1.
        This is a text description for the DCWA GUI's first part.
%%
3
GUI file part 2.
        This is a text description for the DCWA GUI's second part.
```

4. A Revised Query may also limit the personal view without changing the attributes the user wishes to have displayed :

```
DISPLAY : Node Name,Description
FOR : Node Id > 1
```

Answer :

```
2
GUI file part1.
        This is a text description for the DCWA GUI's first part.
%%
3
GUI file part 2.
        This is a text description for the DCWA GUI's second part.
```

5. If a Query (in this case a New Query) does not list any attributes to be displayed then only the node id(s) will be returned :

```
DISPLAY :
FOR : Last Edit Day == Wed
```

Answer :

```
1
%%
7
```

6. If a Query (in this case a New Query) results in an empty hit list then a message with an empty data field will be returned to the UI without consideration of what is in the "display part" of the query :

```
DISPLAY : Node Name,Creator
FOR : Node Id > 7
```

   Answer :

7. If a Revised Query occurs after a query which produced an empty hit list then again a message with an empty data field will be sent to the UI since a further refinement of nothing is nothing :

```
DISPLAY : Node Name
FOR : Node Id == 0
```

   Answer :

8. This New Query shows how the user-defined attribute/value pairs may be used in a search :

```
DISPLAY : Node Name,Description
FOR : new_attribute2 == new_value2
```

   Answer :

```
0
Our files.
        This is a text description for the DCWA.
```

9. This New Query results in an empty hit list since no node has a user-defined attribute of "my_attribute" :

```
DISPLAY : Node Name,Description
FOR : my_attribute == apples
```

Answer :

10. This New Query results in an empty hit list since even though a node [0] has a user-defined attribute of "new_attribute1", its associated value is not "hello" :

```
DISPLAY : Node Name,Description
FOR : new_attribute1 == hello
```

Answer :

11. The next four Queries (all New Queries) show that the creation and last edit times should be split up into day, date, and/or time by the user :

```
DISPLAY : Creation Day,Creation Date
FOR : Last Editor == ShefaliGajiwala
```

Answer :

```
1
Mon
Jun 27 1994
%%
2
Wed
Jun 29 1994
%%
3
Wed
Jun 29 1994
```

12. DISPLAY : Node Name,Topic
    FOR : Last Edit Date >= Aug 25, 1994

    Answer :

```
3
GUI file part 2.
dcwa gui
%%
5
DB file part1.
dcwa database
%%
7
DB file part3.
dcwa database
```

13. `DISPLAY : Node Name,Topic`
    `FOR : Last Edit Day == Sat`

    Answer :

    ```
    3
    GUI file part 2.
    dcwa gui
    %%
    6
    DB file part2.
    dcwa database
    ```

14. `DISPLAY : Node Name,Creator`
    `FOR : Last Edit Time > 15:00:00`

    Answer :

    ```
    0
    Our files.
    KaiChang
    %%
    2
    GUI file part1.
    ShefaliGajiwala
    ```

```
%%
5
DB file part1.
WesleyWear
%%
6
DB file part2.
WesleyWear
```

9.4 Semantic Network

This example shows how a semantic network about automobile safety may be overlaid upon a logical structure concerning the components of the car. The logical view tree used in this example is as follows :

```
0 ____
      |
      | - 1 ____
      |         | - 2
      |         |
      |         | - 3
      |         |
      |         | - 4
      | - 5
      |
      | - 6 ____
      |         | - 7
      |         |
      |         | - 8 ____
      |                   | - 9
      |                   |
      |                   | - 10
      | - 11 ___
                | - 12
                |
                | - 13
```

Assume that the logical view file contains the following information for the node id and the user-defined attribute/value pairs :

```
0
Make Model Year
Generic_Motors XV-1 1995
%{
1
Automobile Safety
Frame Impact_Studies
%{
2
Impact_Studies
Sides
%%
3
Impact_Studies
Rear
%%
4
Impact_Studies
Front
%}
5
Automobile Safety
Engine Engine_TBA
%%
6
Automobile
Interior
%{
7
Interior Safety
Seats Seatbelts
%%
8
```

```
Interior Safety
Dash Airbags
%{
9
Airbags
Driver_Side
%%
10
Airbags
Passenger_Side
%}
%}
11
Automobile
Wheels
%{
12
Wheels Safety
Tires Recommended_Tires
%%
13
Wheels Safety
Brakes Anti-lock_Brakes
%}
%}
```

First, these user-defined attribute/value pairs indicate that :

- The brakes and tires are part of the wheel assembly.
- The seats and dash are part of the interior.
- The frame, engine, interior, and wheels are part of the automobile.
- The specific car in question is Generic Motors' 1995 XV-1.

The above semantic information clearly follows the logical structure, however another semantic structure (concerning the car's safety) may also be examined :

```
Safety ____
           |
           | - Airbags ____
           |              | - Driver Side
           |              |
           |              | - Passenger Side
           |
           | - Anti-lock Brakes
           |
           | - Engine Safety is To Be Accomplished
           |
           | - Impact Studies ____
           |                    | - Front
           |                    |
           |                    | - Rear
           |                    |
           |                    | - Sides
           |
           | - Recommended Tires
           |
           | - Seatbelts
```

The following are sample queries showing a possible search of the safety information :

1. First the user makes a New Query to determine some of the attribute/value pairs that the group has established :

   ```
   DISPLAY : User Attribute-Value Pairs
   FOR : Node Id > 11
   ```

   Answer :

   ```
   12
   Wheels Safety
   Tires Recommended_Tires
   ```

```
%%
13
Wheels Safety
Brakes Anti-lock_Brakes
```

2. Seeing that "Safety" is one of the user-defined attributes, the user makes a New Query to determine if the car has Airbags :

   ```
   DISPLAY : User Attribute-Value Pairs
   FOR : Safety == Airbags
   ```

   Answer :

   ```
   8
   Interior Safety
   Dash Airbags
   ```

   The user may then want to read the detailed information about the car's airbags. However, node 8 contains no information since it is not a leaf node. A user familiar with DCWA would know this and a new user of DCWA would be informed of this condition if he/she tried to pick that node for reading. In either case, the user may make the following New Query.

3. ```
   DISPLAY : User Attribute-Value Pairs
   FOR : Node Level == LEAF
   ```

   Answer :

   ```
   2
   Impact_Studies
   Sides
   %%
   3
   Impact_Studies
   Rear
   %%
   4
   ```
   ```

```
Impact_Studies
Front
%%
5
Automobile Safety
Engine Engine_TBA
%%
7
Interior Safety
Seats Seatbelts
%%
9
Airbags
Driver_Side
%%
10
Airbags
Passenger_Side
%%
12
Wheels Safety
Tires Recommended_Tires
%%
13
Wheels Safety
Brakes Anti-lock_Brakes
```

4. To determine which leaf nodes concern "Airbags", the user would then make the following Revised Query :

```
DISPLAY : User Attribute-Value Pairs
FOR : User Attributes == Airbags
```

Answer :

```
9
Airbags
Driver_Side
%%
10
Airbags
Passenger_Side
```

Thus, nodes 9 and 10 are leaf nodes whose content concerns Airbags and the user may then want to read these two nodes.

5. Finally, since the user is also concerned about other interior safety features of the car, he/she makes the following New Query :

```
DISPLAY : User Attribute-Value Pairs
FOR : User Attributes == Interior
```

Answer :

```
7
Interior Safety
Seats Seatbelts
%%
8
Interior Safety
Dash Airbags
```

Note that node 6 does not meet this criteria since for that node "Interior" is the *value* of the user-defined "Automobile" attribute. The user may then want to read node 7, which is another leaf node concerning the interior safety of the car.

10 Conclusion

The CSCW software tool known as the DCWA, will be useful to any group of people who have a need to coauthor documents, especially if they are separated geographically. This includes, but is not limited to, such diverse fields as scientific research, education, administration, international business, software engineering, fiction and non-fiction writing, screenplay editing, and office report writing. The DCWA can help users cooperate on any writing task logically, conveniently, and efficiently.

One of the reasons people work together is because it is usually easier to divide a problem or task into parts. Not only is this easier, it is usually the only practical way when dealing with today's often large and complex problems. The DCWA will allow the users to increase their information exchange and project coordination, thus enhancing their productivity.

The DCWA provides both textual and graphic editing, viewing, and coordination services to the group members. The design of the tool follows the spirit of "division of labor" through the logical structure's locking mechanism. However, the "shared mind" approach can also be realized by viewing any other member's work space, as well as querying the node attributes and semantic network.

The experience gained from the design and implementation of the prototype helped to contribute to the design of the second version, especially in the design of the second version's database. As stated before, the main challenges associated with designing a database for CSCW are :

- Implementing the partitioning method which will break the documents into manageable sections.
- Maintaining this partitioned structure as changes are made by each user.
- Updating the master copy as needed.
- Determining if two users are trying to modify the same section.

The solutions to these problems as implemented in the DCWA's database are :

- The partitioning method used is the *logical view* defined by the group. The database does not impose a structure upon the group but rather is flexible enough to accommodate any structure. Note that it unites multiple media into one homogeneous structure.

- The users may modify this structure and the changes are reflected at each site.
- The users may make changes to a node and save them locally. Finally, they commit the changes to update the master copy.
- A locking mechanism was implemented to prevent two users from modifying the same node. The logical view's use of non-overlapping work spaces guarantees a conflict-free collaborative environment. Sharing of the nodes is possible without such an activity being completely transparent to the users. In fact, any user may determine the node(s) which another group member is currently working on.

The second version's database also provides the following features. The first two are standard database capabilities made available in a CSCW tool for the first time and the other three are completely novel features.

- Queries within a CSCW environment.
- Security based upon flexible node access permissions.
- Dynamic modification of the logical structure.
- Node attributes, including user-defined attribute/value pairs.
- An overlaid Semantic Network.

The distributed, multimedia database organizes both the textual and graphical information according to their structural relationships, as well as their semantical relationships. Users may navigate through the files logically by using the search facility, which queries the node attributes and limits the user's view to only those nodes which meet the criteria of the query.

The prototype's database provided the minimum necessary requirements and helped prove that the sharing of a single file in real-time was possible within the given networking environment, thus laying the groundwork for a complete multimedia, CSCW software package. The second version's database greatly expanded upon the prototype's database and should become an important platform for future software designers interested in a distributed database for multiuser, multidocument, multimedia, CSCW applications.

References

[1] Saul Greenberg. *Computer Supported Cooperative Work and Groupware*, chapter 1, pages 1–12. Academic Press, Ltd. London, 1991.

[2] Kaj Gronbaek, Morten Kyng, and Preben Mogensen. "CSCW challenges in large-scale technical projects - a case study". In *Proc. of the 1992 CSCW Conference*, pages 338–345, Nov. 1992.

[3] J. Cuena and A. Garcia-Serrano. "Intelligent Computer Support". *Cooperation among Organizations — The Potential of Computer Supported Cooperative Work*, pages 72–102, Power, R.J.D. (Ed.), Germany: Springer-Verlag, 1993.

[4] R. J. D. (Ed.) Power. *Cooperation among Organizations — The Potential of Computer Supported Cooperative Work*, chapter 2,6, pages 13–25,103–118. Germany: Springer-Verlag, 1993.

[5] D. Diaper. "Small-Scale Collaborative Writing Using Electronic Mail". *CSCW in Practice: An Introduction and Case Studies*, pages 72–102, Diaper and C. Sanger (Eds.), Germany: Springer-Verlag, 1993.

[6] B. Hewitt and G. N. Gilbert. "Groupware Interface". *CSCW in Practice: An Introduction and Case Studies*, pages 31–38, Diaper and C. Sanger (Eds.), Germany: Springer-Verlag, 1993.

[7] M. Sharples. "Adding a Little Structure to Collaborative Writing". *CSCW in Practice: An Introduction and Case Studies*, pages 51–68, Diaper and C. Sanger (Eds.), Germany: Springer-Verlag, 1993.

[8] J. N. Orr. "Graphics and Groupware: Increasing Intimacy through Broadening Bandwidth". In *Groupware'92*, pages 73–76, Morgan Kaufmann Publishers, 1992.

[9] J. Grudin. "Computer-Supported Cooperative Work: History and Focus". *IEEE Computer*, pages 19–26, May 1994.

[10] Dimitri Bertsekas and Robert Gallager. *Data Networks*, chapter 1, pages 1–35. Prentice Hall, Inc., Englewood Cliffs, NJ, 1992.

[11] Tom Rodden and Gordon Blair. "CSCW and Distributed Systems : The Problem of Control". In *Proc. of the Second European Conference on CSCW*, pages 49–64, Amsterdam, The Netherlands, 1991.

[12] Mark Roseman and Saul Greenberg. "GROUPKIT : A Groupware Toolkit for Building Real-Time Conferencing Applications". In *Proc. of the 1992 CSCW Conference*, pages 43–50, Nov. 1992.

[13] P. V. Rangan. "Managing Multimedia Collaboration". In *Groupware'92*, pages 418–420, Morgan Kaufmann Publishers, 1992.

[14] T. W. Malone, K. R. Grant, and F. A. Turbak. "The Information Lens: an Intelligent System for Information Sharing in Organization". In *Proc. of the ACM Human Factors in Computing Systems Conference*, pages 1–8, Boston, MA, 1986.

[15] T. W. Malone and K. Lai. "Object Lens: A Spreadsheet for Cooperative Work". In *Proc. of the Conference on Computer Supported Cooperative Work*, Portland, OR, 1988.

[16] T. Rodden. "Technological Support for Cooperation". *CSCW in Practice: An Introduction and Case Studies*, pages 1–22, 1993.

[17] J. Brooke. "User Interface for CSCW Systems". *CSCW in Practice: An Introduction and Case Studies*, pages 72–102, Diaper and C. Sanger (Eds.), Germany: Springer-Verlag, 1993.

[18] Stephen Viller. "The Group Facilitator : A CSCW Perspective". In *Proc. of the Second European Conference on CSCW*, pages 81–95, Amsterdam, The Netherlands, 1991.

[19] Shoshana Loeb. "Delivering Interactive Multimedia Documents over Networks". *IEEE Communications Magazine* , pages 52–59, May 1992.

[20] J. M. Haake and B. Wilson. "Supporting Collaborative Writing of Hyperdocuments in SEPIA". In *Proc. of the 1992 CSCW Conference*, pages 138–146, Nov. 1992.

[21] Mark F. Riley, James J. Feenan, John L. Janosik, and T.K. Rengarajan. "The Design of Multimedia Object Support in DEC Rdb". *Digital Technical Journal* , 5(2):50–64, Spring 1993.

[22] Dimitris N. Chorafas and Heinrich Steinmann. *Solutions for Networked Databases : How to Move from Heterogeneous Structures to Federated Concepts*, chapter 5, pages 73–92. Academic Press, Inc. San Diego, 1993.

[23] K. Watabe, S. Sakada, K. Maeno, H. Fukuoka, and T. Ohomri. "Distributed Multiparty Desktop Conferencing: MERMAID". In *Proc. of the Conference on the Computer Support Cooperative Work*, Los Angeles, CA, 1990.

[24] L. G. Palmer and R. S. Palmer. "DECSPIN: A Networked Desktop Videoconferencing Application". *Digital Technical Journal*, Vol. 5(No. 2):pp.65–76., Spring 1993.

[25] M. Stefik, G. Foster, D. Babrow, K. Kahn, S. Lanning, and L. Suchman. "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving In Meeting". *Communications of the ACM*, Vol.30(No.1):pp.32–47, Jan. 1987.

[26] J. Conklin. "Hypertext: An Introduction and Survey". *Computer, 20(9), IEEE*, pages 17–20; 32–41, September 1987.

[27] R. S. Fish, R. E. Kraut, M.D. Leland, and M. Cohen. "Quilt: A Collaborative Tool for Cooperative Writing". In *Proc. of the Conference on Office Information Systems*, pages 30–37, Palo Alto, CA, 1988.

[28] S. Opper. "A Groupware Tool Box". *BYTE*, pages 275–282, Dec. 1988.

[29] G. L. Rein and C. A. Ellis. "rIBIS: A Real-Time Group Hypertext System". *Int. Journal of Man-Machine Studies*, pages 339–367, Aug. 1993.

[30] I.R. Campbell-Grant. "Introducing ODA". *Computer Standards & Interfaces* , 11(3):149–157, 1991.

[31] Richard Carr. "ODA - The ISO Standard for Electronic Document Interchange". *Computer Standards & Interfaces* , 7(3):297–301, 1988.

[32] Wolfgang Appelt, Richard Carr, and Nikolas Tetteh-Lartey. "Conformance Testing of ODA Documents and ODA Implementations According to ISO 8613 Based on its Formal Description (FODA)". *Computer Standards & Interfaces* , 11(1):47–55, 1990.

[33] J. Lee. "SIBYL: A Tool for Managing Design Rationale". In *Proc. of the Conference on the Computer Support Cooperative Work*, Los Angeles, CA, 1990.

[34] Roger S. Pressman. *Software Engineering : A Practitioner's Approach*, chapter 1, pages 29–30. McGraw-Hill, Inc., 1992.

[35] W. Richard Stevens. *UNIX Network Programming*, chapter 3, pages 88–93. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

[36] Narayana Vuppula. "A Graphical User Interface Design for Computer Supported Cooperative Work". Technical report, Auburn University, Dept. of Computer Science, 1994.

[37] Asad R. Khan. "Real-Time Computer Supported Cooperative Work - The Network Module". Technical report, Auburn University, Dept. of Computer Science, 1994.

[38] The DCWA Working Group. "RFC 1 : A Specification of the DCWA Prototype Version 2". Technical report, Auburn University, Dept. of Computer Science, 1994.

[39] Bjarne Stroustrup. *The C++ Programming Language*, chapter 1-7,R, pages 13–254,477–632. Addison-Wesley Publishing Co., Reading, MA, 1991.

[40] R. C. Schank. *Conceptual Information Processing*, chapter 3, pages 22–82. Amsterdam: North Holland, 1975.

[41] Sun Microsystems. *Network Programming Guide*, chapter 1, pages 4–5. Sun Microsystems, Inc., 1990.

[42] User Services Working Group. "FYI on Questions and Answers". Technical report, Internet Engineering Task Force, 1990.

[43] K. H. Chang, Y. Gong, A. R. Khan, R. Vuppula, and A. W. Wear. "DCWA : A Distributed Collaborative Writing Aid". Technical report, Auburn University, Dept. of Computer Science, 1994.

[44] Kai H. Chang, Yu Gong, Tim Dollar, Shefali Gajiwala, Byong Lee, and A. Wesley Wear. "On Computer Supported Collaborative Writing Tools for Distributed Environments". Technical report, Auburn University, Dept. of Computer Science, 1994.

A Glossary : DCWA Terminology and Abbreviations

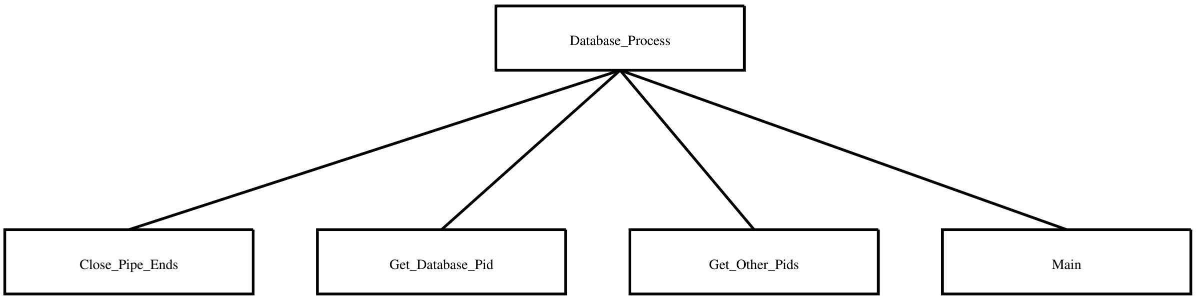
- Active Member - A group member who is currently running DCWA.
- CR - The controller process of the DCWA which forks off the DB, NA, and UI processes.
- CSCW - Computer-Supported Cooperative Work.
- DB - The database process of the DCWA.
- DB Client - A local database which processes messages generated by the UI in response to user actions. Actions which may cause a conflict are passed to the DB Server, otherwise the message is processed locally.
- DB Server - The database which acts as a central database for a group during a DCWA working session. It is responsible for preventing conflicts.
- DCWA - The Distributed Collaborative Writing Aid.
- Group - People who coauthor a set of documents.
- Inactive Member - A group member who is not currently running DCWA.
- Logical View - A general, acyclic, unweighted graph used to maintain the nodes in an organized structure.
- Network File Server - "a facility for sharing files in a heterogeneous environment of machines, operating systems, and networks. Sharing is accomplished by mounting a remote file system, then reading or writing files in place. The goal of the NFS design was to make all disks available as needed. Individual workstations have access to all information residing anywhere on the network." [41]
- Node - A node of the Logical View defines a sharable object within DCWA. A leaf node represents a part of a file as defined by the group, whereas non-leaf nodes connect the various parts of the Logical View tree together.
- Node Attributes - A set of properties which a node possesses.
- NA - The network access process of the DCWA.
- Protocol - "A formal description of message formats and the rules two computers [or processes] must follow to exchange those messages." [42]

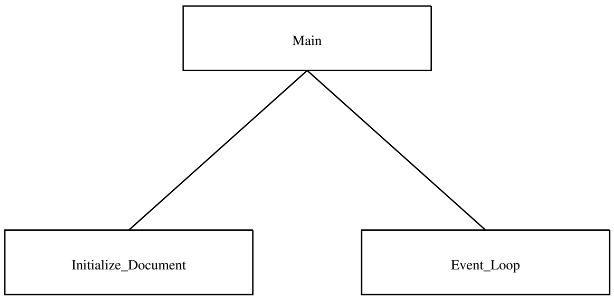
- Session - The time from when a user starts DCWA, being the only active member of the group, until the last active member of the group quits DCWA.
- UI - The user interface process of the DCWA. Although this interface is "graphical" and the usual abbreviation is GUI, UI is used so that this process has a two letter abbreviation like the other DCWA processes.

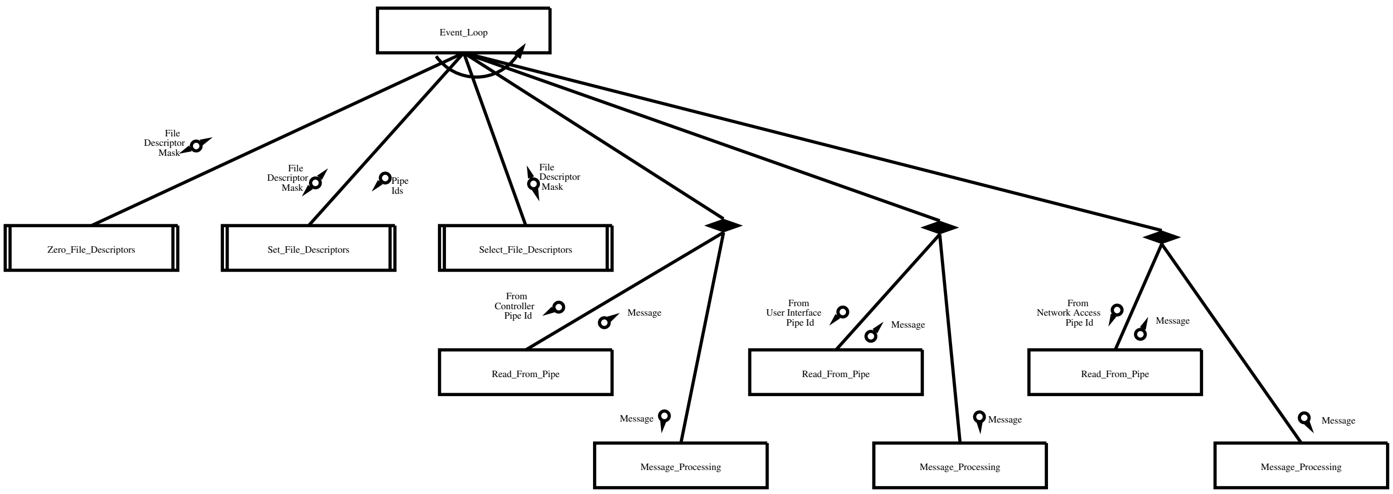
B Time Table

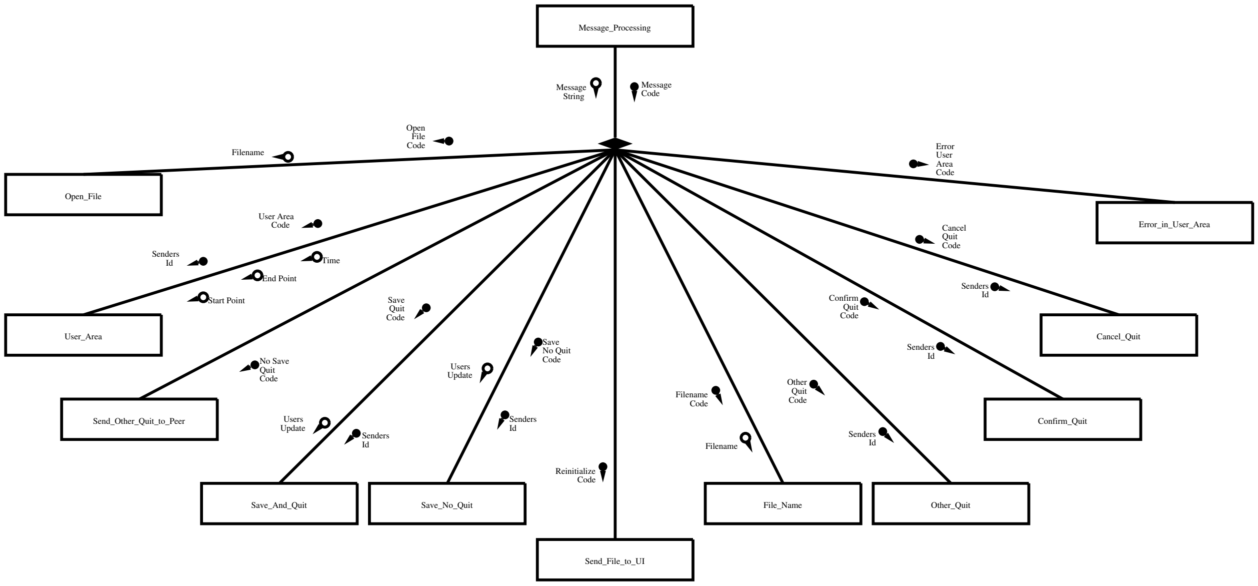
- Spring '93 – The Author joined the DCWA Working Group.
- End of Summer '93 – Completed the prototype database.
- End of Fall '93 – Completed the testing and debugging of the prototype.
- Winter '94 – Participated in a demonstration of the prototype to a group from Gunter Air Force Base.
- Winter '94 – Advisors submitted a paper entitled *DCWA : A Distributed Collaborative Writing Aid* [43] to a CSCW Conference but it was rejected.
- Spring '94 – Completed the specification of the second version of the database.
- Summer '94 – Completed encoding the second version of the database.
- End of Summer '94 – Advisors submitted a paper entitled *On Computer Supported Collaborative Writing Tools for Distributed Environments* [44] to ACM CSC '95 and it was accepted.
- Mid-quarter Fall '94 – Complete the testing and debugging of the second version of the database.

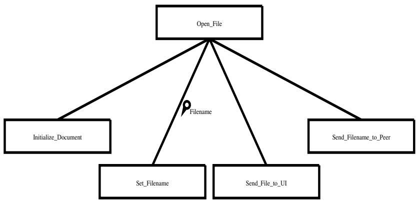
C Structure Charts for the Prototype Database

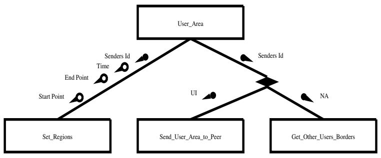


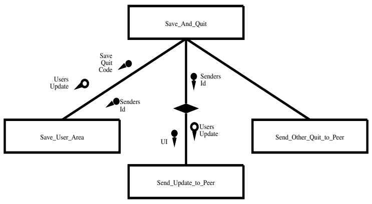


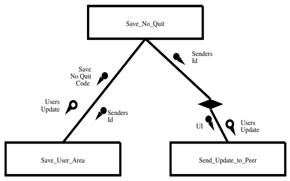


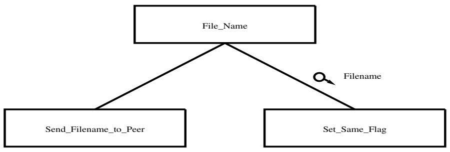

Message_Processing
Message String
Message Code
Open File Code
Filename
Error User Area Code
Open_File
Error_in_User_Area
User Area Code
Cancel Quit Code
Senders Id
Time
End Point
Start Point
Senders Id
Confirm Quit Code
Save Quit Code
User_Area
Cancel_Quit
Save No Quit Code
Senders Id
No Save Quit Code
Users Update
Send_Other_Quit_to_Peer
Confirm_Quit
Filename Code
Other Quit Code
Users Update
Senders Id
Senders Id
Senders Id
Filename
Reinitialize Code
Save_And_Quit
Save_No_Quit
File_Name
Other_Quit
Send_File_to_UI

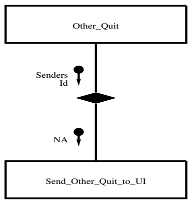

Open_File
Filename
Initialize_Document
Send_Filename_to_Peer
Set_Filename
Send_File_to_UI

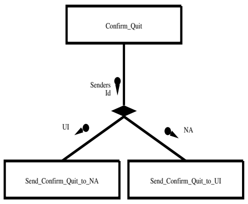

User_Area
Senders Id
Time
End Point
Start Point
Senders Id
UI
NA
Set_Regions
Send_User_Area_to_Peer
Get_Other_Users_Borders

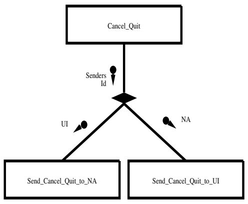

Save_And_Quit
Save Quit Code
Users Update
Senders Id
Senders Id
UI
Users Update
Save_User_Area
Send_Other_Quit_to_Peer
Send_Update_to_Peer

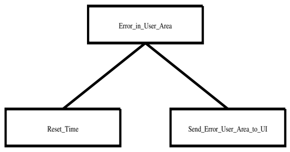

Save_No_Quit
Senders Id
Save No Quit Code
Users Update
Senders Id
UI
Users Update
Save_User_Area
Send_Update_to_Peer

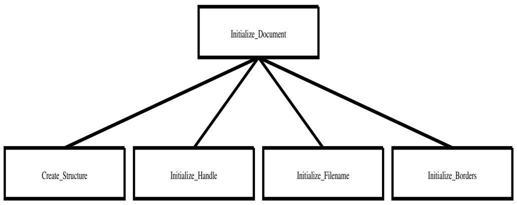

File_Name
Filename
Send_Filename_to_Peer
Set_Same_Flag

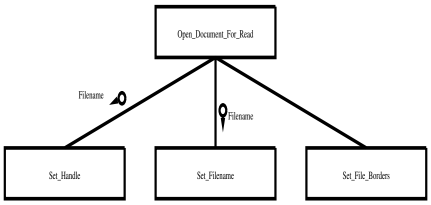

Other_Quit
Senders Id
NA
Send_Other_Quit_to_UI

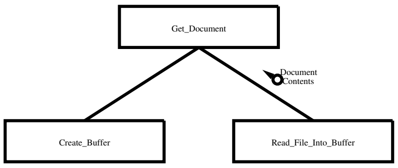

Confirm_Quit
Senders Id
UI
NA
Send_Confirm_Quit_to_NA
Send_Confirm_Quit_to_UI

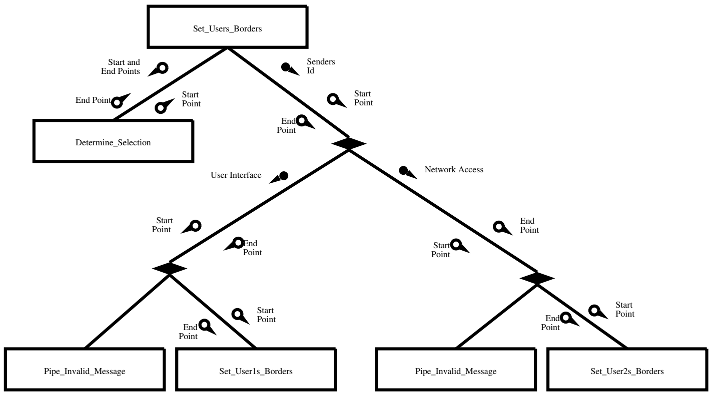

Cancel_Quit
Senders Id
UI
NA
Send_Cancel_Quit_to_NA
Send_Cancel_Quit_to_UI

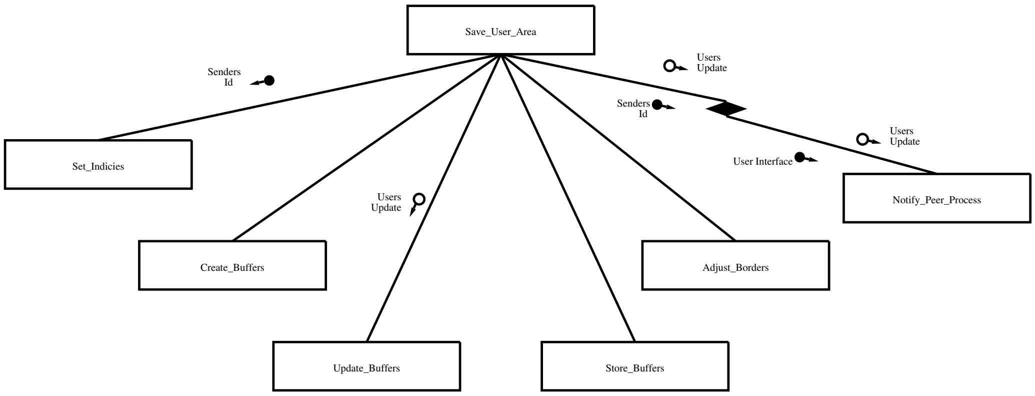

Error_in_User_Area
Reset_Time
Send_Error_User_Area_to_UI


Initialize_Document
Create_Structure
Initialize_Handle
Initialize_Filename
Initialize_Borders


Open_Document_For_Read
Filename
Filename
Set_Handle
Set_Filename
Set_File_Borders


Get_Document
Document Contents
Create_Buffer
Read_File_Into_Buffer


Set_Users_Borders
Start and End Points
End Point
Start Point
Senders Id
Start Point
End Point
Determine_Selection
User Interface
Network Access
Start Point
End Point
Start Point
End Point
End Point
Start Point
End Point
Start Point
Pipe_Invalid_Message
Set_User1s_Borders
Pipe_Invalid_Message
Set_User2s_Borders


Save_User_Area
Senders Id
Users Update
Senders Id
Users Update
User Interface
Set_Indicies
Notify_Peer_Process
Users Update
Create_Buffers
Adjust_Borders
Update_Buffers
Store_Buffers

D Protocols for the Prototype Database

When the Database receives a message with a certain code it performs the actions listed below it. UI is the User Interface process, DB is the Database process, NA is the Network Access process, and N_A means not applicable.

```
1) OPEN_FILE :
           i) Initialize the document variable in case the user is starting
              work on a new file.
          ii) Store the new filename.
         iii) Send the contents of the file to the UI.
          iv) Send the filename to the peer process via the NA.

2) USER_AREA :
           i) Set up the various regions of the document.
          ii) If the sender is UI and the sameflag is TRUE
                 Send the message to the peer process via the NA.
         iii) If the sender is NA and the sameflag is TRUE
                 Send the border points of the other user's selected area to
                 the UI.

3) NOSAVE_QUIT :
        If the sameflag is not N_A
                 Inform the peer process via the NA that the user wants
                 to quit.

4) SAVE_QUIT :
           i) Save the user's changes.
          ii) If the sender is UI and the sameflag is TRUE
                 Send the message to the peer process via the NA.
         iii) If the sameflag is not N_A
                 Inform the peer process via the NA that the user wants
                 to quit.
```

5) SAVE_NOQUIT :

  i) Save the user's changes.

  ii) If the sender is UI and the sameflag is TRUE

    Send the message to the peer process via the NA.

6) RE_INIT :

  Send the contents of the file to the UI.

7) FILENAME :

  If the sameflag is N_A and the filename is not NULL

    Send the filename to the peer process via the NA.

    Set the sameflag depending upon the two filenames.

8) OTHER_QUIT :

  If the sender is NA

    Send the message to the UI.

9) CONFIRM_QUIT :

  If the sender is UI

    Send the message to the NA.

  Else if the sender is NA

    Send the message to the UI.

  Exit DB.

10) CANCEL_QUIT :

  If the sender is UI

    Send the message to the NA.

  Else if the sender is NA

    Send the message to the UI.

11) ERR_USER_AREA :

  i) Reset the time the user picked an area to N_A.

  ii) Relay this message to the UI.

12) Other messages are ignored by the DB.

E Protocols for the Second Version's Database

The following demonstrate the DCWA protocols that are relevant to the database. (See the DCWA_Message structure at the end of the section on the modules.) Note that some "NACKs" below may also have an err_reason of OUT_OF_MEMORY.

0a) A new DB_Client starts up and MAX_USERS has not been reached.

```
        From DB_Client to DB_Server :
                DBC_DBS_NEW_DB_CLIENT

        From DB_Server to DB_Client :
                DBS_DBC_DB_CLIENT_ID, id

        From DB_Server to all other DB_Clients :
                DBS_DBC_NEW_DB_CLIENT, id, data = username machinename
```

0b) A new DB_Client starts up but MAX_USERS has been reached.

```
        From DB_Client to DB_Server :
                DBC_DBS_NEW_DB_CLIENT

        From DB_Server to DB_Client :
                DBS_DBC_DB_CLIENT_ID, id = MAX_USERS_REACHED

        From DB_Client to UI :
                MAX_USERS_REACHED and err_reason = MAX_USERS_REACHED also

        From DB_Client to NA :
                DB_NA_QUIT
```

0c) A new DB_Client starts up and has not yet received an id from the DB_Server but receives some message from the CR, UI, or NA.

```
        From DB_Client to CR, UI, or NA :
                DB_UNINIT
```

0d) A new DB_Client starts up just as the DB_Server is quitting. (The DB_Server did not migrate to the new DB_Client because it did not know about it in time.) However, the NA will soon find out that its successor and predecessor are itself and will inform the DB, which will instantiate a DB_Server object.

```
        From NA to DB :
                NA_DB_FIRSTHOST
```

0e) The DB_Server upon start up (or a DB_Client at the time of receiving an id from the DB_Server) found some DCWA files in the /tmp directory (presumably still there because of a host failure.) The DB informs the UI of this fact. The DB cannot do more than this since, even though the node_id is associated with the file, a group member may have changed the logical view since the file(s) were writen to disk. Therefore, the user should look at the files and determine what to do with them.

```
        From DB_Client to UI :
                DB_UI_ERROR_RECOVERY
```

1a) The user wants to create a new logical view.

```
        From UI to DB_Client :
                UI_DB_NEW_LV

        From DB_Client to UI :
                DB_UI_NEW_LV_ACK
```

```
1b) The user wants to create a new logical view but an error occurred.

        From UI to DB_Client :
                UI_DB_NEW_LV

        From DB_Client to UI :
                DB_UI_NEW_LV_NACK, err_reason one of :

                        OUT_OF_MEMORY


2a) The user wants to save the changes made to the new logical view. (If
the user is also dividing the contents of preexisting files into certain
leaf nodes, the UI should keep that information until the user commits
the new logical view.)

        From UI to DB_Client :
                UI_DB_SAVE, data = structure of the new logical view

        From DB_Client to UI :
                DB_UI_SAVE_ACK


2b) The user wants to save the changes made to the new logical view but an
error occurred.

        From UI to DB_Client :
                UI_DB_SAVE, data = structure of the new logical view

        From DB_Client to UI :
                DB_UI_SAVE_NACK, err_reason one of :

                        CANT_OPEN_TMP_FOR_WRITE
```

3a) The user wants to undo the changes made to the new logical view.

```
From UI to DB_Client :
        UI_DB_UNDO

From DB_Client to UI :
        DB_UI_UNDO_ACK, data = version of the structure
        of the new logical view that is on top of the
        "undo" stack
```

3b) The user wants to undo the changes made to the new logical view but an error occurred.

```
From UI to DB_Client :
        UI_DB_UNDO

From DB_Client to UI :
        DB_UI_UNDO_NACK, err_reason one of :

                CANT_OPEN_TMP_FOR_READ
```

4a) The user wants to undo the undo (redo) made to the new logical view.

```
From UI to DB_Client :
        UI_DB_REDO

From DB_Client to UI :
        DB_UI_REDO_ACK, data = version of the structure
        of the new logical view that is on top of the
        "redo" stack
```

4b) The user wants to undo the undo (redo) made to the new logical view but an error occurred.

```
        From UI to DB_Client :
                UI_DB_REDO

        From DB_Client to UI :
                DB_UI_REDO_NACK, err_reason one of :

                        REDO_STACK_EMPTY
                        CANT_OPEN_TMP_FOR_READ
```

5a) The user wants to commit the new logical view under a filename. The last saved version (the one on top of the "undo" stack) will be committed.

```
        From UI to DB_Client :
                UI_DB_COMMIT_LV_AS, data = filename

        From DB_Client to UI :
                DB_UI_COMMIT_LV_AS_ACK
```

5b) The user wants to commit the new logical view under a filename but an error occurred.

```
        From UI to DB_Client :
                UI_DB_COMMIT_LV_AS, data = filename

        From DB_Client to UI :
                DB_UI_COMMIT_LV_AS_NACK, and err_reason one of :

                        CANT_OPEN_TMP_FOR_READ
                        INVALID_LV
                        CANT_OPEN_LV_FOR_WRITE
```

5c) The user wants to commit the new logical view under a filename and also commit the division of the contents of preexisting file(s) into certain leaf nodes. The UI should send a UI_DB_COMMIT_LV_AS message as per 5a then send the following messages to commit the contents of the leaf nodes. Each leaf node's contents will be committed to disk under the filename that was specified for that node in the logical view. Sending UI_DB_COMMIT_LEAF messages for empty leaf nodes is not required. If only one leaf node is non-empty, only a UI_DB_COMMIT_LAST_LEAF message need be sent (after the UI_DB_COMMIT_LV_LEAVES message). It is assumed that the leaf nodes will be sent in a top-down fashion.

```
        From UI to DB_Client :
                UI_DB_COMMIT_LV_LEAVES, data = filename of logical view

        From UI to DB_Client :
                UI_DB_COMMIT_LEAF, node_id, data = contents

        ... (As many other UI_DB_COMMIT_LEAF messages as needed.)

        From UI to DB_Client :
                UI_DB_COMMIT_LAST_LEAF, node_id, data = contents

        From DB_Client to UI :
                DB_UI_COMMIT_LV_LEAVES_ACK
```

5d) The user wants to commit the new logical view under a filename and also commit the division of the contents of preexisting files into certain leaf nodes but an error occurred. If the error involves committing the logical view itself, the protocol will be the same as 5b. Otherwise, the following messages are sent.

```
        From UI to DB_Client :
                UI_DB_COMMIT_LV_LEAVES, data = filename of logical view

        From DB_Client to UI :
```

```
        DB_UI_COMMIT_LV_LEAVES_NACK, data = filename of logical
        view and err_reason one of :

                CANT_OPEN_LV_FOR_READ
                CANT_OPEN_LV_FOR_WRITE

or

    From UI to DB_Client :
        UI_DB_COMMIT_LEAF or UI_DB_COMMIT_LAST_LEAF, node_id,
        data = contents

    From DB_Client to UI :
        DB_UI_COMMIT_LV_LEAVES_NACK, node_id, data = node's
        filename, and err_reason one of :

                CANT_OPEN_FOR_APPEND


6) The user wants to quit the new logical view without commiting.

    From UI to DB_Client :
        UI_DB_QUIT_LV


7a) The user wants to open an existing logical view file.

    From UI to DB_Client :
        UI_DB_OPEN_LV, data = filename

    From DB_Client to DB_Server :
        DBC_DBS_OPEN_LV, data = filename

    From DB_Server to DB_Client :
        DBS_DBC_OPEN_LV_ACK, data = logical_view
```

```
        From DB_Client to UI :
                DB_UI_OPEN_LV_ACK, data = part_of_logical_view (node id,
                node name, and parent node's name)
```

7b) The user wants to open an existing logical view file but an error occurred.

```
        From UI to DB_Client :
                UI_DB_OPEN_LV, data = filename

        From DB_Client to DB_Server :
                DBC_DBS_OPEN_LV, data = filename

        From DB_Server to DB_Client :
                DBS_DBC_OPEN_LV_NACK, data = filename, and err_reason
                one of :

                        CANT_OPEN_LV_FOR_READ
                        EMPTY_LV
                        INVALID_LV
                        NOT_CURRENT_GROUP_LV

        From DB_Client to UI :
                        DB_UI_OPEN_LV_NACK, data = filename, and err_reason
                        as above.
```

7c) The UI needs the logical view again. The DB assumes a logical view file has already been opened. If one has not, a default logical view of one node will be passed back to the UI.

```
        From UI to DB_Client :
                UI_DB_RESEND_LV

        From DB_Client to UI :
```

DB_UI_RESEND_LV_ACK, data = part_of_logical_view (node id, node name, and parent node's name)

8) The user wants to quit (close) the logical view.

From UI to DB_Client :
UI_DB_QUIT_LV

From DB_Client to DB_Server :
DBC_DBS_QUIT_LV

9a) The user wants to query the database about (the nodes of) the logical view.

From UI to DB_Client :
UI_DB_NEW_QUERY, data = query

From DB_Client to UI :
DB_UI_QUERY_ACK, data = user-specified fields of the nodes which fit the criterion.

9b) The user wants to revise the query based on what was obtained from the previous query.

From UI to DB_Client :
UI_DB_REVISED_QUERY, data = query

From DB_Client to UI :
DB_UI_QUERY_ACK, data = user-specified fields of the nodes which fit all the old criteria, as well as the new criteria.

9c) There is no DB_UI_QUERY_NACK since it is assumed that the UI will be able to prevent invalid query formation because the user must select items from a list rather than type in the entire query directly.

10a) The UI needs to know which group member is editing a node.

```
        From UI to DB_Client :
                UI_DB_EDITOR_INFO, node_id

        From DB_Client to UI :
                DB_UI_EDITOR_INFO_ACK, node_id, and
                data = username machinename
```

10b) The UI needs to know which group member is editing a node but an error occurred.

```
        From UI to DB_Client :
                UI_DB_EDITOR_INFO, node_id

        From DB_Client to UI :
                DB_UI_EDITOR_INFO_NACK, node_id, and err_reason one of :

                        NO_SUCH_NODE
                        NO_CURRENT_EDITOR
```

11a) The UI needs to know about the attributes of a node.

```
        From UI to DB_Client :
                UI_DB_NODE_ATTRS, node_id

        From DB_Client to UI :
                DB_UI_NODE_ATTRS_ACK, node_id and data = all node attributes
```

11b) The UI needs to know about the attributes of a node but an error occurred.

```
        From UI to DB_Client :
                UI_DB_NODE_ATTRS, node_id

        From DB_Client to UI :
                DB_UI_NODE_ATTRS_NACK, node_id and err_reason one of :

                        NO_SUCH_NODE
```

12a) The user wants to read a node.

```
        From UI to DB_Client :
                UI_DB_READ, node_id

        From DB_Client to UI :
                DB_UI_READ_ACK, node_id, and data = contents_of_node
```

12b) The user wants to read a node but an error occurred.

```
        From UI to DB_Client :
                UI_DB_READ, node_id

        From DB_Client to UI :
                DB_UI_READ_NACK, node_id and err_reason one of :

                        NO_SUCH_NODE
                        PERMISSION_DENIED
                        EMPTY_NODE
```

13) The user has finished reading a node.

```
        From UI to DB_Client :
                UI_DB_QUIT_READ, node_id
```

14a) The user wants to edit a node.

```
        From UI to DB_Client :
                UI_DB_EDIT, node_id

        From DB_Client to DB_Server :
                DBC_DBS_EDIT, node_id

        From DB_Server to DB_Client :
                DBS_DBC_EDIT_ACK, node_id, and data = contents_of_node

        From DB_Server to all other DB_Clients :
                DBS_DBC_NEW_EDITOR_ID, node_id, data = user_id

        From DB_Client to UI :
                DB_UI_EDIT_ACK, node_id, and data = contents_of_node
```

14b) The user wants to edit a node but an error occurred.

```
        From UI to DB_Client :
                UI_DB_EDIT, node_id

        From DB_Client to UI :
                DBC_DBS_EDIT, node_id

        From DB_Server to DB_Client :
                DBS_DBC_EDIT_NACK, node_id and err_reason one of :

                        NO_SUCH_NODE
```

```
                    PERMISSION_DENIED
                    ALREADY_LOCKED_FOR_EDIT
                    ALREADY_LOCKED_FOR_LV_MOD
                    DESCENDANT_LOCKED_FOR_EDIT
                    DESCENDANT_LOCKED_FOR_LV_MOD

        From DB_Client to UI :
                DB_UI_EDIT_NACK, node_id and err_reason as above or :

                    OUT_OF_MEMORY

                in which case the DB_Client informs the DB_Server of the
                local error :

        From DB_Client to DB_Server :
                DBC_DBS_FREE_NODE

        From DB_Server to all other DB_Clients :
                DBS_DBC_NEW_EDITOR_ID, node_id, data = N_A


15a) The user wants to save the changes made to the node.

        From UI to DB_Client :
                UI_DB_SAVE, node_id, data = new_contents

        From DB_Client to UI :
                DB_UI_SAVE_ACK, node_id


15b) The user wants to save the changes made to the node but an error
occurred.

        From UI to DB_Client :
                UI_DB_SAVE, node_id, data = new_contents
```

From DB_Client to UI :
DB_UI_SAVE_NACK, node_id, and err_reason one of :

CANT_OPEN_TMP_FOR_WRITE

16a) The user wants to undo the changes made to the node.

From UI to DB_Client :
UI_DB_UNDO, node_id

From DB_Client to UI :
DB_UI_UNDO_ACK, node_id, data = old_contents

16b) The user wants to undo the changes made to the node but an error occurred.

From UI to DB_Client :
UI_DB_UNDO, node_id

From DB_Client to UI :
DB_UI_UNDO_NACK, node_id, and err_reason one of :

CANT_OPEN_TMP_FOR_READ

17a) The user wants to undo the undo (redo).

From UI to DB_Client :
UI_DB_REDO, node_id

From DB_Client to UI :
DB_UI_REDO_ACK, node_id, data = old_contents

17b) The user wants to undo the undo (redo) but an error occurred.

```
From UI to DB_Client :
        UI_DB_REDO, node_id

From DB_Client to UI :
        DB_UI_REDO_NACK, node_id, and err_reason one of :

                REDO_STACK_EMPTY
                CANT_OPEN_TMP_FOR_READ
```

18a) The user wants to commit the changes made to the node. The version on top of the "undo" stack will be used, so any final changes should be saved before committed (i.e., the UI should send a UI_DB_SAVE message as per 15a above.)

```
From UI to DB_Client :
        UI_DB_COMMIT, node_id

From DB_Client to DB_Server :
        DBC_DBS_COMMIT, node_id, data = contents

From DB_Server to DB_Client :
        DBS_DBC_COMMIT_ACK, node_id

From DB_Server to all other DB_Clients :
        DBS_DBC_NEW_EDITOR_ID, node_id, data = N_A

From DB_Client to UI :
        DB_UI_COMMIT_ACK, node_id
```

18b) The user wants to commit the changes made to the node but an error occurred.

```
From UI to DB_Client :
        UI_DB_COMMIT, node_id

From DB_Client to DB_Server :
        DBC_DBS_COMMIT, node_id, data = contents

From DB_Server to DB_Client :
        DBS_DBC_COMMIT_NACK, node_id, and err_reason one of :

                CANT_OPEN_LV_FOR_WRITE
                NOT_EDITOR

From DB_Client to UI :
        DB_UI_COMMIT_NACK node_id, and err_reason as above.
```

19) The user wants to quit the node without commiting.

```
From UI to DB_Client :
        UI_DB_QUIT_NODE

From DB_Client to DB_Server :
        DBC_DBS_FREE_NODE

From DB_Server to all other DB_Clients :
        DBS_DBC_NEW_EDITOR_ID, node_id, data = N_A
```

20a) The user wants to modify (a subtree of) the logical view, starting at node_id. If node_id == 0, then the request is to lock the entire logical view for modification. A modification message can also be used to modify node attributes, not just adding or deleting nodes.

```
From UI to DB_Client :
        UI_DB_MODIFY_LV, node_id

From DB_Client to DB_Server :
        DBC_DBS_MODIFY_LV, node_id

From DB_Server to DB_Client :
        DBS_DBC_MODIFY_LV_ACK, node_id

From DB_Server to all other DB_Clients :
        DBS_DBC_NEW_EDITOR_ID, node_id, data = user_id

From DB_Client to UI :
        DB_UI_MODIFY_LV_ACK, node_id
```

20b) The user wants to modify (a subtree of) the logical view, starting at node_id but an error occurred.

```
From UI to DB_Client :
        UI_DB_MODIFY_LV, node_id

From DB_Client to UI :
        DBC_DBS_MODIFY_LV, node_id

From DB_Server to DB_Client :
        DBS_DBC_MODIFY_LV_NACK, node_id and err_reason one of :

                NO_SUCH_NODE
                PERMISSION_DENIED
                ALREADY_LOCKED_FOR_EDIT
                ALREADY_LOCKED_FOR_LV_MOD
                DESCENDANT_LOCKED_FOR_EDIT
                DESCENDANT_LOCKED_FOR_LV_MOD

From DB_Client to UI :
```

DB_UI_MODIFY_LV_NACK, node_id and err_reason as above or :

OUT_OF_MEMORY

in which case the DB_Client informs the DB_Server of the local error :

From DB_Client to DB_Server :
DBC_DBS_FREE_NODE

From DB_Server to all other DB_Clients :
DBS_DBC_NEW_EDITOR_ID, node_id, data = N_A

21a & 21b) The user may save the changes made to (a subtree of) the logical view. The protocols are the same as 15a & 15b.

22a & 22b) The user may undo the changes made to (a subtree of) the logical view. The protocols are the same as 16a & 16b.

23a & 23b) The user may redo the changes made to (a subtree of) the logical view. The protocols are the same as 17a & 17b.

24a & 24b) The user may commit the changes made to (a subtree of) the logical view. The protocols are the same as 18a & 18b, except 24a will also include :

From DB_Server to all other DB_Clients :
DBS_DBC_UPDATED_LV, node_id,
data = new logical view (subtree)

From DB_Client to UI :

```
                DB_UI_UPDATED_LV, node_id,
                data = new logical view (subtree)
```

and 24b will not use NOT_EDITOR but NOT_LV_MODIFIER.

25) The user may quit the logical view modification without commiting. The protocol is the same as 19.

26a) The user with a client database wants to quit. In these protocols, the UI quits without waiting on the DB or NA to quit. They may continue sending messages for awhile before quitting. Also, the quit is not acknowledged since the UI has already quit.

```
        From UI to DB_Client :
                UI_DB_QUIT

        From DB_Client to DB_Server :
                DBC_DBS_QUIT

        From DB_Server to DB_Client :
                DBS_DBC_QUIT_ACK

        From DB_Server to all other DB_Clients :
                DBS_DBC_DB_CLIENT_QUITTING, data = user_id

        From DB to NA :
                DB_NA_QUIT
```

26b) The user with the server database wants to quit and no other group members are still running DCWA. Of course, the UI and NA do not know that their sibling DB is the DB_Server, so "DB (Server)" is used below instead of DB_Client. (The DB_Server acts as a db client to its local siblings; it is only a db server to other DBs.)

```
From UI to DB (Server) :
        UI_DB_QUIT

From DB (Server) to NA :
        DB_NA_QUIT
```

26c) The user with the server database wants to quit but other group members are still running DCWA.

```
From UI to DB (Server) :
        UI_DB_QUIT

From DB_Server to DB_Client chosen as the new DB_Server :
        DBS_DBC_NEW_DB_SERVER

From chosen DB_Client to DB_Server :
        DBC_DBS_DB_SERVER_ACK

From old DB_Server to new DB_Server :
        DBS_DBS_DB_SERVER_INFO, data = info

From old DB_Server to NA :
        DB_NA_QUIT
```

26d) The user with the server database wants to quit, other group members are still running DCWA but the chosen DB_Client's user decides to quit before the DBS_DBC_NEW_DB_SERVER message arrives.

```
From UI to DB (Server) :
        UI_DB_QUIT

From (chosen host's) UI to (chosen) DB_Client :
        UI_DB_QUIT
```

From DB_Server to chosen DB_Client :
DBS_DBC_NEW_DB_SERVER

From chosen DB_Client to DB_Server :
DBC_DBS_DB_SERVER_NACK

DB_Server must then choose another client to become the server and more messages are sent out as before.

26e) The user with the server database wants to quit, other group members are still running DCWA but the chosen DB_Client's user decides to quit after the DBC_DBS_DB_SERVER_ACK message has already been sent.

From UI to DB (Server) :
UI_DB_QUIT

From DB_Server to chosen DB_Client :
DBS_DBC_NEW_DB_SERVER

From chosen DB_Client to DB_Server :
DBC_DBS_DB_SERVER_ACK

From (new DB_Server's) UI to new DB_Server :
UI_DB_QUIT

From old DB_Server to new DB_Server :
DBS_DBS_DB_SERVER_INFO, data = info

From old DB_Server to NA :
DB_NA_QUIT

From new DB_Server to (another) chosen DB_Client :
DBS_DBC_NEW_DB_SERVER

Then messages continue as before.

26f) If a message from a DB_Client or a local process comes to the new DB_Server before the old DB_Server has sent the db server info, then the new DB_Server will store the message in a queue and will process all such queued messages when the DBS_DBS_DB_SERVER_INFO message is received.

F Suggested Recommendations for Future Work

The following is a list, in no particular order, of recommended improvements to anyone who wishes to continue this work :

For the DB :

1. Incorporate other media, such as audio and video communication, into the logical view.
2. Catch and process any signals which the DB process may receive, except of course SIGKILL, which may not be caught.
3. Conduct timing experiments to determine if there is an advantage or disadvantage to being the user whose database also happens to be the database server.
4. Use the effective gid feature of UNIX for the group's files.
5. Allow dynamic additions/removals from the group by using dynamic memory allocation rather than the array of size MAX_USERS to hold the ids.
6. Make the database robust with respect to host (local and remote) failures.
7. Create a simulator so that the database may be run/tested independently of the other DCWA processes.
8. Use Flex/Bison to scan/parse the logical view 1) when reading it in from a file, and 2) when committing a user's modifications to a new or existing logical view. Is there a way to scan a string and not an input file? Can more than one scanner/parser pair be in a program? With respect to the last question, can the various `yyparse()` functions be renamed or overloaded? Will pointers to functions help? Also, note that Flex and Bison were not used in the second version of the database since neither the `CC` nor `g++` compilers would compile a program that was compilable under the `gcc` compiler.
9. Allow a user to create, store, open and close various personal views per DCWA editing session. This can be accomplished by maintaining one "absolute" logical view and many relative logical views. Within each relative (personal) logical view, each node would have an absolute and relative node id. You might would want to start by still limiting each group to one "absolute" logical view per session. Would an "absolute" logical view modification access list be needed?

10. Collaboration among the group will usually span more than a single session, so it is desirable to maintain session state information over the full duration of the project. This means that certain objects will need to be persistent from session to session by writing them to disk. Currently, only the logical view and the files represented in it are persistent. Is there any other "group data" which needs to be stored from one session to another?

11. Create different classes for the different types of nodes (logical, text, graphics, etc.) derived from some base class. Watch out for the parent pointer field.

12. Maintain a "history list" of editors and commit times rather than just the last editor and commit time. Associated with this would be keeping the various versions of the node available. (Or have a command-line switch to turn this feature on and off.) Who should be allowed to delete this list except for the most recent entry? Who should be allowed to delete the old versions of the node?

13. Allow a user to modify an existing logical view even if it is not the group's current logical view. Do not let the user start editing nodes since the DB Server only maintains one logical view per group per session right now. The modification to the logical view that is not currently being used by the group would involve only the local database.

14. Allow nodes to become any size by using the LongStringCell and LongString classes everywhere. This means the other DCWA processes would have to use them too.

15. Currently, any user that may modify the logical view starting at some node may also modify that node's user-specified attributes. But these include the access lists themselves. Should another mechanism be used to change these access lists? Which user(s) should be allowed to define them and/or change them? What about their hierarchical nature with respect to this new mechanism?

16. Include a new access list attribute (called comment_access) which holds the list of people who may comment upon the node. That is, they are more than passive readers of the node but they may not actually edit the node either, just post comments. Of course, a method for storing (and eventually deleting) these comments must be provided.

17. Once the group finishes editing the documents, they may wish to allow outsiders to read (and/or comment – see above) their work from within DCWA, so that the search facility is available, for example. Therefore, a mechanism that allows anyone to read

a leaf node should be provided so that the group can make their work available to the world.

18. At the other extreme from the above, the group may want to close a node completely.

19. Get rid of "logical" nodes and let each non-leaf node access the entire contents of its descendants. This will, of course, require the UI to be able to display the various media together, as per number 2 below.

20. Add "wildcards" to the query mechanism. For the values this makes sense but what about wildcards for the attributes' names too?

For the UI :

1. Conduct Human Computer Interaction (HCI) tests on the UI. Note that most HCI research has been directed towards single-user systems [1], so a good project or thesis could come from such a study.

2. Create a new widget that can handle text, graphics, hypertext links, multiple fonts, audio, video, etc.

3. Have different icons in the display of the logical view for different types of nodes : logical, text, graphics, audio, etc.

For DCWA :

1. Get rid of all the pipes (and possibly the NA process) and have the DB and UI use sockets.

2. Unless another use for the pids is found, get rid of the CR sending each process the process ids of its sibling processes which were forked off after it. This is a holdover from when signals were going to be used to send messages between the processes before pipes and the select system call were implemented.

3. Maintain a fully connected socket network between the nodes, rather than a ring topology, yet keep the concept of successor and predecessor. Thus, "broadcasted" messages can go around the ring but messages to a specific host can go directly with minimal delay.