TECHNICAL REPORT

OBJECT SHARING IN A MULTI-USER HYPERTEXT SYSTEM

M.S. Thesis

Melvina H. Tarazi

CCS TR # 101

CENTER FOR COORDINATION SCIENCE

Massachusetts Institute of Technology
Sloan School of Management
Cambridge, Massachusetts
OBJECT SHARING IN A MULTI-USER HYPERTEXT SYSTEM

M.S. Thesis

Melvina H. Tarazi

CCS TR # 101
OBJECT SHARING IN A MULTI-USER HYPERTEXT SYSTEM

By

Melvina H. Tarazi

S.B. Computer Science and Engineering
Massachusetts Institute of Technology
(1988)

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1989

Copyright (c) 1989 - Melvina H. Tarazi

The author hereby grants MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author

Department of Electrical Engineering and Computer Science
June 20, 1989

Certified by

Professor Thomas W. Malone
Thesis Supervisor

Accepted by

Professor Arthur C. Smith
Chairman, Committee on Graduate Students
OBJECT SHARING IN A MULTI-USER HYPERTEXT SYSTEM

by

Melvina H. Tarazi

Submitted to the Department of Electrical Engineering and Computer Science on June 20, 1996 in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

Object-oriented databases and hypertext systems are emerging in response to a demand by application developers for increased, less structured modeling power in database systems. Such systems succeed in providing the necessary tools and facilities for building cooperative work applications but are limited in providing the appropriate object sharing environment.

We propose to address the object sharing requirements for one such system - Object Lens. Object Lens integrates features of hypertext systems, object-oriented systems and rule-based agents.

We evaluate various approaches to object sharing (including message passing, centralized object server and distributed object servers) and various schemes for concurrency control and update propagation (locking, timestamping) with respect to the characteristics desired in Object Lens (e.g., specialization hierarchy, user-interface, object linking, version control and combination of long interactive transactions and short automatic transactions).

We propose a new scheme for initiating object sharing through the exchange of electronic mail messages. Object protection is achieved by a hybrid scheme of access control lists and capability systems.

Analysis of the transactions in Object Lens reveals two sets of transactions: (1) interactive transactions that require relatively weak consistency requirements and relatively flexible concurrency control and, (2) automatic transactions that require strict concurrency control requirements. We propose a hybrid locking and version control concurrency control scheme that accommodates the two types of transactions.

Thesis Supervisor: Prof. Thomas W. Malone
Title: Patrick J. McGovern Professor of Information Systems
Dedication

To My Parents
Acknowledgements

I would like to thank my thesis supervisor, Professor Thomas Malone, for his insight, guidance, and support. His ideas, observations, and suggestions have deeply contributed to the success of this work.

I would also like to thank Mr. Kevin Crowston for reading several drafts of this work. His suggestions, critiques and comments were extremely helpful.

My parents, Reine and Hani, deserve more than I can express for being there when I needed them, and for all their love, affection, guidance and encouragement throughout my life. May this thesis be a partial compensation to them for being away from home.

Special thanks go to my brother, Ramzi, for listening to my gripes and complaints throughout my stay at M.I.T.

Finally, I would like to acknowledge Ibrahim Saad, Nazhin Zarghamee, Ala Alryyes, Eman Hashem and Ahmad Tabari for their constant friendship throughout my stay at M.I.T.
Table of Contents

Abstract
Dedication
Acknowledgements
Table of Contents
List of Figures
List of Tables

1. Introduction
   1.1 Object Sharing in Object-Oriented Databases and Hyperext Systems
   1.2 The Proposed Object Sharing Scheme
   1.3 Object Lens
   1.4 Thesis Approach
   1.5 Thesis Outline

2. Architectures for Object Sharing
   2.1 Message Based Object Sharing
      2.1.1 Advantages of Approach
      2.1.2 Disadvantages of Approach
   2.2 Centralized Object Sharing
      2.2.1 Advantages of Approach
      2.2.2 Disadvantages of Approach
      2.2.3 Improved Performance using Caching
      2.2.4 Implementation of Central Data Server
         2.2.4.1 Central Database Server
         2.2.4.2 Central Object Server
   2.3 Distributed Object Sharing
      2.3.1 Advantages of Approach
      2.3.2 Disadvantages of Approach
      2.3.3 Implementation of a Distributed Data Server
         2.3.3.1 Distributed Object Server
         2.3.3.2 Object Server and Manager at each Workstation
      2.3.3.3 Distributed Relational Database Systems

3. Object Sharing Features in a Distributed Environment
   3.1 Replication
   3.2 Consistency and Update Propagation
   3.3 Concurrency Control
      3.3.1 Synchronization Techniques Based on Two-Phase Locking
         3.3.1.1 Basic 2PL Implementation
         3.3.1.2 Primary Copy 2PL
         3.3.1.3 Voting 2PL
         3.3.1.4 Centralized 2PL
         3.3.1.5 Granularity of Locking
         3.3.1.6 Deadlock Detection
      3.3.2 Synchronization Techniques Based on Timestamp Ordering
3.3.2 Basic Timestamp Ordering
3.3.2.2 Multi-Version Timestamp Ordering
3.3.2.3 Conservative Timestamp Ordering
3.3.3 Synchronization Techniques Based on Optimistic Control
  3.3.3.1 Majority Consensus
  3.3.3.2 Distributed Certification
3.3.4 Comparison of the Three Methodologies
3.3.5 Other Concurrency Control Schemes

3.4 Security and Protection
  3.4.1 Capability Systems
  3.4.2 Access Control List System

4. Literature Survey
  4.1 Distributed Relational Database Systems
  4.2 Object-Oriented Database Systems
  4.3 Multi-user Hyper-text Systems

5. Object Lens Revisited
  5.1 Object Lens as a Hyper-text System
  5.2 Object Lens as an Object-Oriented Database System
  5.3 Object Lens Specific Features
  5.4 Using Object Lens
  5.5 Object Lens Architecture
  5.6 An example

6. Multi-user Object Lens
  6.1 Early Design Decisions
  6.2 Terminology
  6.3 Objects in Distributed Object Lens
    6.3.1 Shared vs. Personal Objects
    6.3.2 Private vs. Public Objects
    6.3.3 Local vs. Remote Objects
  6.4 Creation of a Shared Object
  6.5 Deletion of a Shared Object
  6.6 Protection of a Shared Object
  6.7 Modification of a Shared Object
    6.7.1 Transactions in Object Lens
      6.7.1.1 Interactive Transactions
      6.7.1.2 Automatic Transactions
    6.7.2 Concurrency Control
      6.7.2.1 Time-Stamp Ordering
      6.7.2.2 Optimistic Concurrency
      6.7.2.3 Two-Phase Locking
      6.7.2.4 Version Control
      6.7.2.5 Proposed Synchronization Technique: Hybrid of Locking and Versioning

7. Conclusion
  7.1 Summary of Work
  7.2 The Proposed Distributed Object Lens
  7.3 Direction of Future Work
List of Figures

Figure 2-1: Centralized Object Server
Figure 2-2: Distributed Object Server
Figure 2-3: Object Server and Manager at each Workstation
Figure 2-4: Distributed Relational Database
Figure 3-1: Example of Conflicts in transactions
Figure 3-2: Deadlock Situation
Figure 3-3: Wait-for Graph
Figure 5-1: Person object in Object Lens
Figure 5-2: Architecture of Object Lens
Figure 6-1: Object Definitions
Figure 6-2: Two-user Distributed Object Lens
Figure 6-3: System Wide Unique Object ID
Figure 6-4: Interactive Transactions
Figure 6-5: Rule execution transactions
Figure 6-6: Interleaving of Rules
Figure 6-7: Resolving Modify-Modify Conflict using Hybrid Implementation
Figure 6-8: Resolving a Read-Modify Conflict using the Hybrid Implementation
Figure 6-9: Rule Execution Deadlock
Figure 7-1: Distributed Object Lens
List of Tables

Table 4.1: Distributed Database Features 54
Table 4.2: Features of Object-Oriented Databases 60
Table 4.3: HyperText Features 65
Chapter 1
Introduction

Conventional record-oriented relational databases are restricted in the kind of information they can manage. They provide data modeling and transaction management capabilities that are well suited to business data processing but which are not adequate for computer-supported cooperative work applications. Such increasingly important applications include computer-aided design, software development, documentation authoring, and artificial intelligence knowledge bases. These applications are interactive in nature and deal with unstructured objects and complex relationships among data, people and schedules.

Object-oriented databases and hypertext systems attempt to provide the platform to support the semantic and data models required for these applications. Object Lens [Lai 88, Malone 89] is one such system. It integrates features of hypertext and object-oriented systems to provide a system that may be used to write specific applications for retrieving and browsing complex information such as meeting scheduling, project management and document authoring.

However, the current implementation of Object Lens fails to support a key feature of cooperative applications: Object Sharing. Object Lens is currently a single-user system. Our goal is to propose an object sharing scheme that accommodates the key characteristics of Object Lens (long interactive transactions combined with short rule-based transactions, complex objects, naive users, and version maintenance).
1.1 Object Sharing in Object-Oriented Databases and Hypertext Systems

Object and information sharing in object-oriented databases and hypertext systems implies:

- Multiple users should be able to access documents and designs created by other users.
- Collaborating users should be able to view modifications made to any of the shared documents or designs (update propagation).
- Different users working concurrently should not be able to interfere with each other's work (concurrency control).

To support object sharing, these systems need to have a concurrency control scheme, an update propagation scheme, and an object protection scheme. Unfortunately, object-oriented databases and hypertext systems cannot simply adopt the concurrency control schemes (such as locking and timestamping) and update propagation schemes proposed and implemented for distributed databases. The major problems are the different nature of the transactions (long vs. short) and the different nature of the objects (complex vs. simple) supported by these systems and traditional database systems. However, these schemes can be used as the basis for more adequate schemes for object-oriented databases and hypertext systems.

Many object-oriented databases prototypes (Purdy 87, Fishman 87, Hornick 87) adopt variations of locking for concurrency control. Some systems use a check-in-check-out protocol to disallow multiple users from accessing an object concurrently. Existing distributed hypertext systems use either locking or version control to resolve possible conflicts from concurrent users. Most of these systems do not support replication (multiple copies of an object), or version maintenance.
1.2 The Proposed Object Sharing Scheme

We argue that the traditional concurrency control schemes as well as their variants proposed for object-oriented databases are not adequate for systems where (1) user friendliness is of major importance, and (2) modification may be made to the objects interactively or automatically.

User friendliness implies that (a) redoing user performed modifications to an object is intolerable, (b) disallowing users to access objects for long periods of time is again intolerable.

We propose a concurrency control scheme that is a hybrid of locking and version control. This scheme makes no restrictions on replication, i.e., objects may be replicated and the scheme would still be applicable. It also accommodates the existence of different versions of an object.

We feel that this scheme is suitable for object-oriented and hypertext applications that emphasize user friendliness and that combine interactive modifications through editing of an object form and automatic modifications through the execution of rule-based transactions.

1.3 Object Lens

Object Lens is an integrated (hypertext, object-oriented database, rule-based) environment for developing cooperative work applications. It combines the formal representation of knowledge (using structures such as frames, inheritance networks and production rules) in knowledge-based systems with the informal representation of information (unstructured text or graphics) in hypertext systems to present a semiformal middleground. Object Lens adds semantic structure to hypertext nodes that allows the ease of summarization of object
contents and relationships and automatic search and manipulation of a collection of objects. It may be used to develop specific applications for information sharing, retrieving and browsing of complex information, meeting scheduling, project management and documentation development (hypertext), by including explicit knowledge on various objects such as different message types, people, task, meetings, products and companies.

The basic primitives available to the programmer to build his applications are:

1. Facilities for creating object instances and object types through a template-based user interface
2. Facilities to modify objects by editing the window display corresponding to the object
3. Facilities to create rule-based agents that automatically process information on behalf of the user
4. Facilities to collect objects together into folders. The folders may be automatically maintained by an agent.

Features and Characteristics of Object Lens

We feel that the following characteristics of Object Lens highly influence our choice of an object sharing scheme:

- **Naïve Users**: Object Lens is geared toward non-programming users. This makes user-interface a major component of the system. The user-interface model influences what aspects of object sharing are exposed to the user.

- **Combination of Automatic and Interactive Modifications of Objects**: Objects may either be modified automatically by "rule-based agents" or interactively by a user editing a display window corresponding to the object.

- **Version Maintenance**: As a tool for implementing cooperative work applications, Object Lens should support the Maintenance of different versions of objects.
Distributed Object Lens

Distributed Object Lens should support object sharing between various users on the network. It should support location transparency, replication transparency, concurrency transparency and possibly access transparency. Users anywhere in the network should be able to reference and dereference objects located elsewhere in the network. It should provide a mechanism for object protection. It should guard against or resolve possible conflicts that arise because of concurrent access to an object by a number of users.

1.4 Thesis Approach

In this thesis, our main concern is exploring the different issues involved in object sharing and the possible solutions to these issues. We are concerned with sharing object “instances” in Object Lens. We will assume all users have the same object hierarchies and object type definitions, i.e. the PERSON object universally means the same thing. We will not be concerned with object type coercion. [Lee 88] explores the various ways of doing type translation.

We explore a large number of distributed systems (distributed relational databases, distributed hypertext systems and distributed object-oriented systems) to examine how object sharing is achieved. Most of these systems are prototypes and experimental. Our emphasis is on the architecture configuration of the distributed systems (centralized or distributed), the concurrency control schemes used, the protection schemes used and the update propagation schemes adopted.

We specify the requirements and features of Distributed Object Lens and then study the suitability of the various techniques for the Object Lens environment.
We finally summarize the tradeoffs between the different techniques in the context of Object Lens and recommend the most appropriate design for Distributed Object Lens.

1.5 Thesis Outline

Chapter 2 examines the various architectures for object sharing in a distributed environment: in the framework of the Server-Client Model.

Chapter 3 describes various features of a distributed system. It surveys the various algorithms and schemes to achieving object sharing in a distributed environment. The relevant features are concurrency control, update propagation and protection.

Chapter 4 presents a literature survey of various distributed systems from relational databases to object-oriented databases to hypertext systems. The emphasis in the survey is on the concurrency control mechanisms adopted, the propagation methods used and the version control schemes explored.

Chapter 5 describes the features and characteristics of Object Lens.

Chapter 6 describes the requirements for Distributed Object Lens. It describes object sharing and protection in the context of the Object Lens environment. It defines the possible transactions in Object Lens and accordingly examines the different techniques for concurrency control describing the tradeoffs. A hybrid scheme best suitable for Distributed Object Lens is proposed.

Chapter 7 summarizes the key tradeoffs between the various choices and schemes and then presents the best appropriate approach for Distributed Object Lens.
Chapter 2

Architectures for Object Sharing

Users of centralized systems such as mainframes are accustomed with the idea of being able to share data and objects with other users of the system. With the advent of personal computers, workstations and networks, the essential schemes and algorithms to support features of object sharing needed to be revised to work in such distributed environments.

The essential features to object sharing in a distributed system are:

1. Allow multiple users at various sites to access objects created by any one user in the system (Shared Access). The access allowed to these objects may differ from one user to another (Read, Write, Delete) (Protection).

2. Multiple copies of an object may exist in the system. The system need to ensure that all copies of the object are consistent (i.e., the same) at any point in time. Consistency is achieved by propagating changes made to one copy of the object to all other copies (Update Propagation).

3. Permit users to access object in a multi-programmed fashion while preserving the illusion that the user is executing on a dedicated system. To attain this goal, the system should prevent modifications made by one user from interfering with reads or updates performed by another (Concurrency Control). The concurrency control problem is exacerbated in a distributed system because: (a) users may simultaneously access objects stored at many different sites in a distributed system and (b) a concurrency control mechanism at one site cannot instantaneously know about interaction at another site.
There are three approaches to object sharing in a distributed environment. The three approaches are message-based, centralized shared object space and distributed shared object space. We focus our attention on the location and maintenance of shared objects. We assume that personal objects are located at the local site for personal use only.

A message-based approach suggests that object sharing could be done through the exchange of specialized electronic mail messages. To a large extent in distributed systems information sharing is done through the use of electronic mail. The suggestion is that this could be extended to incorporate exchange and sharing of objects.

The centralized approach suggests that shared objects be located at a central site in the network. Individual workstations need to direct shared object access to the central site. This approach is highly dependent on the availability of the central site.

The distributed approach suggests that shared objects be distributed at different sites in the network. Users at different sites should be able to access objects located at each site.

We present the centralized and distributed approach to object sharing in framework of a Server-Client Model. The server provides the required object management and objects. The client is locate at each workstation to direct object requests. In the centralized approach, one object server exists in the distributed system. Object caches may be provided at the client site to improve object access. In the distributed approach several object servers exist. These may be dedicated machines as in the central approach, in which case object caches at the client workstation may improve performance. Another option is to locate both the server and the client processes at the workstation. This eliminates the need for a cache.

We examine in this chapter the various possible implementations of each of these schemes in the context of an object-oriented, knowledge-based, hypertext systems similar to Object Lens. Our survey is a synthesis of approaches adopted by different systems. The major features to keep in mind about these systems are:
The next chapter is reserved to examining various algorithms and schemes proposed for concurrency control, update propagation and protection. Chapter 4 presents a survey of distributed systems indicating what object sharing approach they use, and the concurrency control and update propagation schemes adopted.

### 2.1 Message Based Object Sharing

This approach suggests that objects are shared between different users by embedding them in electronic mail messages that are then mailed through the network. The objects could be either literally embedded into the message or embedded in link form. The first method requires that the receiving end extract the object description and the unique object identifier supplied by the underlying system, and build a local copy of the object with the appropriate unique object id. The second method implies that the sender inserts only a reference in the message to the object he wishes to share. It is up to the receiving end to dereference the object by explicitly asking for it if deemed useful. The question is once the object is mailed, who is responsible for maintaining consistency of the different object copies.

Section 4.3 gives a clearer account of this method.

#### 2.1.1 Advantages of Approach

The advantages of the message-based approach are the following:

- Fits well with distributed culture
For systems that are geared towards naive users and that make extensive use of messages to share information, using messages to share objects seems to be a natural evolution. Moreover, naive users may feel more comfortable with explicit (exposed) sharing of objects than with transparent sharing.

2.1.2 Disadvantages of Approach

The disadvantages of the message-based approach are the following:

- Limited to initiating object sharing

This approach mainly focuses on initiating object sharing, i.e., making an object accessible to others. It does not address the maintenance of the shared object. Once there are several copies of the object at different sites, the problem arises of maintaining and updating these objects. Thus evokes into the distributed approach to object sharing.

2.2 Centralized Object Sharing

This approach dictates that all information (objects) deemed sharable, (i.e., accessible by more than 1 person has access to the objects) must be located at a central dedicated site in the network. The overall configuration is depicted in Figure 2-1.

This configuration falls into the Server-Client Model, where the central site acts as a server of shared objects and the other workstations in the network act as clients requesting objects from the server. Users will remotely access objects at the central site from their workstation. The system hides the details of the remote access which is slower than local accesses. Indicating that an access is remote may be useful to explain to the user at the client the reason for the delay. With the increased speed of local area networks the delay would probably be due to processing at end sites.
2.2.1 Advantages of Approach

The advantages of the centralized approach are:

- Simple concurrency control.
- No overhead due to update propagation.
- Simple Rule Resolution

Simple concurrency control

All accesses to shared objects must go through the central server. It would be easy for the central server to control and coordinate concurrent access to the same object. The choice of a mechanism varies from locking (checkout, checkin), timestamping, optimistic control or version merging. Chapter 3 and Chapter 4 discuss the various mechanisms in a great amount of detail. Of interest in this approach are the mechanisms that pertain to a single copy of the object.

No update propagation overhead

All updates are directed to the central server. Only a single accessible copy of a shared object exists and is always obtained from the central site. Hence, any updated propagation overhead encountered in the distributed or cached approach is completely avoided.
Simple Rule Resolution

Simple queries can be formulated in knowledge-based systems using IF-THEN rules. The central approach allows the rules pertaining to shared objects to be resolved at the central site by looking for objects that match and perform the appropriate manipulation.

2.2.2 Disadvantages of Approach

The disadvantages of the centralized approach are the following:

- Contention at Central Site
- Availability of Central Site
- Remote access at every shared object access

Contention at Central Site

The central site needs to process object requests from all workstations in the network. This may lead to a significant degradation in performance. The server should be able to asynchronously accept object requests and pass the object ids to the disk manager for retrieval. Once the object is available it would be returned to the requester client.

Availability of Central Site

The entire system’s access to shared objects is dependent on the availability of one central site. In case of central site failure, users at various workstations will be unable to access any shared objects. This is not a very desirable feature.

Remote access at every shared object access

Every time a shared object is accessed at any workstation this results in a request being sent to the central server demanding the object. Delays are unacceptable when the shared object accessed has not been modified since the last access.
2.2.3 Improved Performance using Caching

This problem of unnecessary successive remote accesses to the same object may be solved by designating an area in memory at the individual workstations to cache remotely retrieved objects. As long as the object has not been modified by other users, its state in the cache is the valid state. Future accesses to the object will be directed to the local cached copy. A write would be done to the cached copy and then written through to the server. Rules and methods may be performed locally on the cached copy.

Granularity of Cache

One issue is at what granularity is caching performed. One simple option is perform caching at the level of an object. Hence, objects are the unit of transfer between the central server and the cache. The other option is to perform caching of larger units such as segments (collection of objects). A basic notion in object-oriented systems is that related objects that would be accessed together should be clustered together. If the segment contained such a cluster of objects, then this approach could lead to improved performance because of a decrease in the number of remote accesses performed, because successive object references are to objects in the same segment.

Validation of Cache

Moreover, the system should guarantee that a user will always access the latest copy. The system should propagate the changes made to an object to all the copies of the object that are cached at workstations. There are number of ways for propagating the updates:

1. The server maintains a list of the objects that are cached at each workstation.
   If an update is received, the list is checked and the appropriate workstations are notified of the update. The client may automatically fetch the updated object or prompt the user to do so. A cleaner implementation would be to set
triggers to check out objects at the server that will automatically trigger an update process if the object is updated.

2. If an object is updated, the server sends messages to the caches that hold a copy of the object invalidating the state of the object. The client attempts to access the object from the cache. If the object is valid it is returned otherwise it is refetched from the server.

3. Timestamps are maintained for objects at the central server. At each access, the timestamp of the object is retrieved from the server and compared to the local timestamp. If different, the object is retrieved. This is faster than retrieving the object as a whole. This requires the maintenance of a larger number of time stamps than previous method. Method 2 required timestamps for segments and used objects in the cache. This method requires timestamps for all objects at the server and in the cache.

Replacement of cached units

The size of the cache is much smaller than the size of memory at the object server. Retrieved objects cannot be cached indefinitely. The cache's limited memory space needs to be managed. An object replacement strategy is required. The options are:

1. LRU (least recently used) may be used to select infrequently accessed objects from the cache.

2. FIFO (first-in-first-out) may be used to select the oldest object in the cache.

3. Random may be used to select the object to be replaced at randomly.

The server needs to be informed of the cache's flushed entries so that it updates its list of
objects and the corresponding caches. If objects are cached our system will have many of the complexities associated with a distributed object server with object replication. Update propagation is one such complexity. Concurrency control will also be more complicated. These are discussed below in the context of distributed object servers.

2.2.4 Implementation of Central Data Server

The next important question is the actual implementation of the central server. There are two main options: a central relational database server and a central object server.

2.2.4.1 Central Database Server

A database server build on top of a conventional relational database may be used to implement a central data server. Shared object instances would be stored in terms of relations. Converting complex objects into a relation is not always a one step task. Difficulties arise because of the network structure of the object space (objects with links to other objects). Put in relational database terminology, it is difficult to normalize the object network. A given object will need to be represented by a number of relations to ensure atomicity of domains and ensure mutual independencies between different attributes. Moreover, converting to normal form makes object (clustering related objects together for fast access) more difficult. Another difficulty arises in mapping semi-structured objects into relations. The contents of a field of a Lens Object could be a combination of text and links. This again is not easy to map into a relation. Mapping hypertext-like objects into relations with pointers to flat files again diminishes the power of hypertext. Most relational database systems do not provide the trigger feature ($R^*$ is an exception\(^1\)). This again is another important feature in the systems we are concerned with. Moreover, since objects are stored as relations at the server methods can not be performed at the server. Methods can only be performed at the client side where objects are formed.

\(^1\)This system is described in Section 4.1
The advantages of using a database server is that it is easily obtainable from the market. It provides the technology for fast data access and concurrency control (although this might not be adequate).

The database server must be able to asynchronously accept requests from various workstations in the network. These requests may be in the form of a set of relational queries that result in the retrieval of the appropriate relations that correspond to the requested object. It is the duty of the client end to translate object requests into the appropriate set of relational queries. Notice that this requires a tremendous amount of processing at both sites. The client end should also extract the necessary information from the returned tuples to build up the requested object.

The communication between the server and the client will be in terms of relational queries and returned tuples. The system will rely on the relational database for transaction management, concurrency control, authorization and recovery.

2.2.4.2 Central Object Server

A central object server may be used to implement the central data server. An object server returns objects that are in the format of objects in object-oriented systems. (Recall that the database server returns tuples that need to be later decomposed at the client side to form the object). Messages between the client and the server would be in the form of object requests similar to those that access local objects. Unique Object IDs or object queries (e.g., description or selection rules) will be used to access objects at the central object server. The central object server will be built on top of a storage manager. The storage manager could either be an extended relational storage system or a persistent object store. The relationship between an object server and objects is analogous to the relationship between a file server and files.
The first approach requires extending a relational storage system to accommodate:

- the different concurrency control requirements that result from the long transactions characteristic of object-oriented hypertext applications.
- version maintenance and control.
- efficiently mapping complex hypertext and semistructured objects into a relational storage system.
- triggering of processes upon the modification of stored objects.

The second approach requires building or acquiring a persistent object store:

- The object store accepts unique object ids or object queries and returns the corresponding objects.
- concurrency control can be customized for the Object Lens environment.
- version maintenance and control.
- version merging.
- recovery and resiliency.
- protection and authorization.
- triggers.

The advantages of the first approach is that a relational storage system may be acquired in the marketplace. Features of fast access, protection and recovery are already provided. The advantages of the second approach is that the object store would be tailored to best suit the applications of interest. The disadvantage is the effort needed to build such an object store.

2.3 Distributed Object Sharing

This approach proposes that shared objects be distributed at various sites in the network.

Objects may be distributed in a manner such that:

1. one copy of the object is present.
2. multiple copies of the object are present (objects are replicated).
3. a copy of the object is replicated at each site (fully redundant).
Replication leads to an improvement in availability and throughput. It also is expensive because it requires (1) increased memory space to hold replicated objects and, (2) increased overhead to ensure that updates go to all copies of an object and that all copies are consistent. Moreover, traditional concurrency control schemes lead to an overall deterioration in performance if replication is performed at more than 4 sites.

In the case of no replication, objects may be allowed to migrate from one site to another. This means that object manipulation may be performed locally.

2.3.1 Advantages of Approach

The advantages of the distributed approach are:
- Increased availability of shared data
- Improved access performance.
- Lessened contention at storage site

**Availability**

In a distributed system access to shared data is no longer dependent on the availability of one site, since the data is distributed over a number of sites. Moreover, replication of data at various sites will make data access insensitive to the failure of a particular site.

**Improved access**

Replication will allow object access to be directed to the closest site that contains a copy of the object. What site this is depends on the implementation. (See Section 2.3.3).

**Lessened contention at single storage site**

In the distributed case, object requests go to one of a number of object servers. This leads to an improvement in performance because object access is no longer restricted to a single machine.
2.3.2 Disadvantages of Approach

The disadvantages of the distributed approach are:
- more complex concurrency control,
- update propagation overhead,
- complex rule resolution,
- complex recovery.

**Complex Concurrency Control**

Allowing object replication complicates concurrency control because concurrent accessers at different sites are harder to detect than concurrent accessers at one site. Each user is able to access the closest copy of an object and perform his changes. The system should be able to detect such conflicts and provide a mechanism for resolving them. Chapter 3 section 3 presents a detailed survey of such detection and resolution mechanisms and Chapter 6 section 4 presents mechanisms appropriate for Distributed Object Lens.

**Update Propagation**

Allowing replication introduces the problem of maintaining consistency between the various copies of an object at the various sites. For traditional database systems a user must be guaranteed that he is reading the latest committed copy of an object. We will see later that this requirement may be relaxed in an environment such as Object Lens. In some cases it is sufficient to give a user access to the latest local copy with an indication that it might not be the latest copy in the distributed system. Chapter 3 section 2 presents a detailed account of the various update propagation strategies and Chapter 6 section 5 describes how propagation of updates may be performed in Distributed Object Lens.

**Complex Recovery**

Recovery and reliability are much more complicated in a distributed system. Network partitions may result in inconsistencies between various copies at various sites. Inconsistencies may be detected but are very hard to resolve.
Complex Rule Resolution

Since objects are distributed over a number of sites, this requires that a rule be applied at the sites where the objects in question are stored in order to search for objects that match the rule. There are three possible ways to handle this:

1. Subrules. The collection of objects being considered may be virtually divided into subcollections indicating their location site. A message is then sent to each of the appropriate site containing the rule and the considered objects. The rules fire at each of the storage sites and the results are directed back to the initiating site. This is analogous at a simpler level to query execution in distributed relational databases.

2. Migration. The required objects may migrate to the site of execution of the rule. The rule is then fired and the appropriate results returned. The migrated objects could be returned to their original site then or at a future time when they are requested.

3. Replication. In the fully redundant case, all objects accessed by a user are already present at his object server. This means that any fired rule does not need to access any remote objects. This of course requires an increase in the amount of memory space needed and an increase in the overhead incurred because of update propagation and complex concurrency control.

2.3.3 Implementation of a Distributed Data Server

There are three main approaches to distributing shared objects over sites in a distributed environment. The sites may be dedicated object servers or extended super workstations that can store local as well as commonly used shared objects. The object-oriented hypertext system may also be implemented on top of a distributed database system. We will address the requirements and implications of each of these approaches in turn.
2.3.3.1 Distributed Object Server

This approach implies the existence of several object servers in the network and might or might not support replication of data. The actual configuration of the network is depicted in Figure 2-2.

![Figure 2-2: Distributed Object Server](image)

In terms of the client-server model, there are a number of dedicated servers in the system. A client service is implemented at each workstation to direct object requests to the relevant server. We will refer to the client as the object manager.

The different object servers collaborate to provide a system-wide shared object space. Such a system should provide location transparency. In other words, the user is not exposed to the location of the object. The system should also provide replication transparency. Objects may be replicated at various object servers for increased performance. Object managers are located at each workstation to direct and control object access to the appropriate object server.

**Duties of Object Manager**

The duties of the object manager can be summarized as follows:

- Locating the requested object.
- Rule or method execution.
- Managing a cache.
Location of object

The object manager needs to locate where the requested object is stored. This may be performed in several ways. One scheme would be to use a name server that given an object unique id, returns the object’s location and internal id. The request is then directed to the appropriate object server. The name server could either be centralized or replicated at each site in the network.

An alternative scheme would be for objects to have unique system-wide object ids that reflect their location. This allows object managers at the different workstations to extract the name of the appropriate object server where the object is located.

If objects are allowed to migrate, then the second approach would not make sense. The name server approach could be used to monitor the change in locations of objects. Every time an object migrates its entry in the name server is updated. If the name server is replicated at each client site, the change in location has to be propagated to all sites. Another possible option is for an object to leave a forwarding address indicating its new location. This becomes slow if objects migrate a lot (since several forwarding addresses need to be followed before the object is located).

Rule or method execution

Rules and methods need to be executed at the objects storage site. It is the responsibility of the object manager to direct the rule execution to the appropriate sites (as explained earlier).

Managing a cache

If caching is going to be used then the object manager’s responsibility is to manage the cache. We mentioned in section 2.2.3 the implications of caching.
Duties of Object Server

The duties of the object server can be summarized as follows:

- Accept object requests or queries and return objects.
- Protection and Authorization of objects.
- Version Control
- Recovery
- Clustering
- Triggers
- Update Propagation
- Concurrency Control

Accept object requests or queries

The object server processes the unique object id and returns the requested object. It should also be able to accept a query indicating specific attributes of an object and return the set of objects that match. Indexing and other features need to be used to improve access time.

Protection and Authorization of Objects

The object manager is responsible for maintaining controlled access to shared objects. Given an object access request and the user's id the object manager verifies that the user is authorized to access the object in the specified mode. Protection may be performed using a capability-based mechanism or access control lists. See Section 3.4 and Section 6.3 for further details.

Version Control

The object manager is responsible for maintaining different versions of the object and alerting the user to the existence of the different versions.

Recovery

The object server is responsible for ensuring the persistence of the objects. It should have a sound recovery scheme in case of site failures.
Clustering

If segments are used as the unit of transfer between the server and the client cache (in case of caching), then the object server is responsible for clustering related objects together into segments stored at the server.

Triggers

The object server should be able to send trigger messages to the object manager if objects have been updated or if links have been added.

Concurrency Control and Update Propagation

It is difficult to classify at what end (object server or object manager) each of these important functionalities fall. It is highly dependent on the concurrency and propagation schemes adopted. Implementation notes are made in Chapter 6 regarding each of the schemes in the context of Object Lens.

2.3.3.2 Object Server and Manager at each Workstation

This approach is an extension of the distributed object server approach. Objects are distributed over the memory space of the workstations. Each workstation is extended to have its local object manager and an object server (already present). Object servers are no longer dedicated machines but implemented at each workstation. Each object server is responsible for local objects that could be shared or personal. Object managers are only responsible for protecting objects at its site. Object access is directed first to the local manager, which in turn directs it to the appropriate object manager if the object is not local. Communication occurs between object managers as opposed to between object manager and object server as in the earlier approach to access an object. Otherwise, the duties of the object manager and the object server are more or less the same as in the distributed object server approach.
2.3.3.3 Distributed Relational Database Systems

This approach suggests that Distributed Object Lens be build on top of a distributed relational database. This again may be acquired in the market place. The distributed relational database is able to provide the data management features and an intermediate interface needs to build to provide the characteristics of an object-oriented system. The distributed relational database provides the location and replication transparency. It will also provide the concurrency control, update propagation, security and authorization. The difficulty arises as in the case of a centralized relational database in mapping object-oriented object space into a relational structure. More over, the concurrency control mechanisms typically provided by distributed databases are not adequate for applications supported by object-oriented database systems or hypertext systems.

To summarize, here are the possible configurations for sharing objects in a distributed environment in the framework of a server-client model.

1. One shared object server / object manager clients at workstations.
   a. Caching
   b. No caching

2. A number of dedicated object servers / object manager clients at workstations.
   a. Replication
      i. Caching
      ii. No caching
   b. No replication
3. Object server and object manager client at each workstation
   a Replication
   b No replication
      i Migration
      ii No migration

The other dimension of this is how the object server is implemented. The two options are:
on top of a relational storage system or on top of an object-oriented storage system. The
requirements of each were outlined.

The difference between a single object server with caching and the distributed approach is
that of concurrency control. Both require update propagation techniques. The former
requires a simple concurrency control since conflicts may be detected at the central site.
Conflicts are harder to detect in the later case, since they need to be detected across server
sites.
Chapter 3

Object Sharing Features in a Distributed Environment

In Chapter 2, we presented two main approaches to object sharing, centralized and distributed. An integral part of both approaches is concurrency control and update propagation. Update propagation is only necessary if more than one copy of an object exists.

In this chapter we attempt to survey different algorithms and schemes proposed to the following features required in a distributed system to provide object sharing capabilities:
- Replication and Update Propagation
- Concurrency Control

We also look at various protection schemes that provide differing access rights to sharers of a shared object.

3.1 Replication

Data replication means that a given logical data can have several distinct stored representatives at several sites. The advantages of such an arrangement are:
- Reduced communication traffic and improved response time by providing local representatives.
- Accesses to the same logical data by several users may be serviced in parallel, improving system throughput.
- Increased system availability by allowing a given data object to be available for processing as long as at least one replica is available.
Supporting replicas, however, complicates the details of data retrieval and modification.

3.2 Consistency and Update Propagation

An update to any given logical data object must be propagated to all stored representatives of the object. Consistency of the representatives must be ensured. Two degrees of consistency are possible:

- Perfect consistency implies that all representatives of the data object have the same initial values and after each modification all the representatives are instantly updated in the same way to remain identical throughout time. This is difficult to ensure because some sites containing the representatives may be unavailable at the time of update.

- Looser forms of consistency imply that all representatives will eventually be updated, so that the representatives converge to the same value at some time interval after updates have stopped.

Update Propagation is a problem even in a single-user distributed system, where concurrency control considerations that arise in a multi-user system may be ignored. There are three basic methods for performing update propagation:

1. **Primary Copy Update.** One representative of the data object is designated as the primary copy. All writes done to the data object must be directed to the primary copy. A write is complete as soon as it has been applied to the primary copy. It is then the primary copy’s site’s responsibility to propagate the update to all other sites that have representatives of the data object. Reads may be directed to any representative of the data object. The disadvantage of
this method is its dependency on the availability of the primary site. One solution to this is the "moving primary site". Once the primary site is down, a new primary site is elected (according to some voting rule).

2. Distributed Update (write-all). A read operation may be directed to any representative of the data object. A write operation is applied to all representatives of the data object. A write is complete only when it has been applied to all representatives. The difficulty arises when some sites are unavailable (disconnected... i.e., a write would fail if not all sites are available to approve the update.

3. Majority Consensus. Again the write are directed to all representatives of an object, however only a majority of sites need to approve the update before the write is committed.

[Demers 88] algorithm allows for a looser consistency control. Background processes ensure that different replicas eventually converge. Hence reads may be directed to an outdated copy of an object. This might be acceptable for some applications.

3.3 Concurrency Control

In a multi-user system, different users may access the same logical data object concurrently. Concurrency control preserves the illusion that each user is executing alone in a dedicated system. Concurrency control prevents data modifications performed by one user from interfering with data retrievals and updates performed by another. The concurrency control problem is exacerbated in a distributed system because:

- Users may access data stored in many different computers in a distributed system.
• Users may concurrently access different representatives of the same logical data object at different sites. One site cannot instantaneously know about interactions at other sites.

Conflicts that may arise fall into two categories: version conflicts and serializable conflicts [Coulouris 88].

Version Conflicts

These result when users access different representatives of an object and perform independent modifications to different fields or slots in the object. Such conflicts may be resolved using version control and merging. To illustrate the notion of version conflicts, consider a file accessed by two concurrent transactions. The first transaction modifies text at the top of a file and the second modifies text at the bottom of a file. The versions can be merged to include both changes. Such concurrency control schemes will be discussed in section 3.3.5.

Serializable Conflicts

These conflicts arise when users access an object concurrently and perform dependent modifications to the object. Here concurrency control is achieved by adopting a synchronization technique that ensures that the outcome of two interleaved transactions is equivalent to execution of the two transactions in serial order [Bernstein 81]. These techniques prevent or detect conflicts when they occur. To illustrate the notion of a serializable conflict, consider a data object \( x \) and two transactions \( T_i \) and \( T_j \). If \( T_i \) issues a read and \( T_j \) issues a write, the value read by \( T_i \) will differ depending on whether the read precedes or follows the write (rw conflict). Similarly if both transactions issue write operations, the value of \( x \) depends on which write happens last (ww conflict). Consider the case where transactions \( T_i \) performs \( x = x + 1 \) and transaction \( T_j \) performs \( x = x - 1 \). Both
transactions need to read and write x. If both transactions are executed concurrently, then the reads and writes may be interleaved in three ways as shown in Figure 3-1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Actual</th>
<th>Intended</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>T</td>
<td></td>
<td></td>
<td>R x 10</td>
<td>W x 11</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>T</td>
<td></td>
<td></td>
<td>R x 10</td>
<td>W x 11</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>T</td>
<td></td>
<td></td>
<td>R x 10</td>
<td>W x 11</td>
</tr>
</tbody>
</table>

Figure 3-1: Example of Conflicts in transactions

Execution (1) is the serialized execution of T₁ and T₂. Execution (2) illustrates a rw and a wv conflict if T₁ and T₂ are concurrent. Execution (3) illustrates a rw and a vw conflict if T₁ and T₂ are concurrent.

Rw conflicts and vw conflicts can lead to anomalies such as a "lost update" or an "inconsistent read". Note that we will see later that serializability might not be required for all transaction in design and hypertext systems. It is, however, of paramount importance in database systems.

Synchronization techniques may be categorized as optimistic or pessimistic in nature.

"Pessimistic" assumes that concurrent transactions accessing the same object will lead to conflict and hence makes one transaction wait for another to complete before it is started. Synchronization techniques based on two-phase locking fall into this category. The disadvantage of this approach is the possibility of deadlock. A deadlock arises when a set of transactions are waiting for each other to commit before they can proceed.

"Optimistic" takes the view that concurrent transactions accessing the same object may not conflict. Transactions are allowed to proceed until a conflict is detected.
techniques based on timestamp ordering and optimistic concurrency control fall into this category. The disadvantage of this approach is the overhead incurred in redoing transactions once conflicts are detected.

The remaining of this section is dedicated to examining the various synchronization techniques that have been proposed. Part of the classification is based on [Bernstein 81].

3.3.1 Synchronization Techniques Based on Two-Phase Locking

Two-phase locking (2PL) synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. Before reading data object \( x \), a transaction must own a \textit{read lock on} \( x \). Before writing into \( x \), it must own a \textit{write lock on} \( x \).

The ownership of locks is governed by two rules:

1. Different transactions cannot simultaneously own conflicting locks; two locks conflict if both are locks on the same data item, one is a \textit{read lock} and the other is a \textit{write lock}, or one is \textit{write lock} and the other is a \textit{write lock}.

2. Once a transaction surrenders ownership of a lock, it may never obtain additional locks.

The first rule describes the first phase (lock acquisition) and the second rule describes the second phase (lock release).

3.3.1.1 Basic 2PL Implementation

\textit{One copy of object}

Associated with each site is a \textit{Lock Manager (LM)} responsible for processing lock requests and releases for objects stored at that site. A transaction wishing to read or write a data object \( x \) requests the corresponding lock from the LM. If the requested lock cannot be granted, the operation is placed on a waiting queue. When a lock is released, the operations on the waiting queue of the object \( x \) are processed in a first-in/first-out order.
Redundant copies of object

This implementation works correctly for redundant copies in the following way: an operation may read any copy and need only obtain a read lock on the copy of \( x \) it actually reads. However, if an operation is updating \( x \), then it must update all copies of \( x \), and so must obtain write locks on all representatives of \( x \). A rw conflict is detected at the site where the read lock is acquired and a ww conflict is detected at all sites. This scheme is vulnerable to global deadlock during the write locks acquisition phase for the different representatives of an object. Two transactions \( T_1 \) and \( T_2 \) might be simultaneously be obtaining write locks on the copies of \( x: x_1, x_2, x_3, x_4, x_5 \). If \( T_1 \) has obtained write locks on \( x_1 \) and \( x_2 \), \( T_2 \) has obtained write locks on \( x_3, x_4 \) and \( x_5 \), neither transactions can proceed any further because neither can obtain locks on the remaining representatives of \( x \). Once the deadlock is detected, one transaction is picked (perhaps the one with fewer locks) and rolled back. The following three locking schemes pay more attention to redundant copies and avoid such lock acquisition deadlock.

3.3.1.2 Primary Copy 2PL

Redundant copies of object:

In this implementation, one copy of the logical data object \( x \) is designated to be the primary copy. Before accessing (read or write) any copy of the logical data object, the appropriate lock must be obtained on the primary copy. For read locks this technique requires more communication than the basic 2PL because the operation needs to communicate with the primary site in order to obtain the lock as well as the site where the read will occur. Both rw and ww conflicts are detected at the primary site.

3.3.1.3 Voting 2PL

Redundant copies of object:

This technique is only suitable for ww synchronization. A transaction lock point occurs when it has obtained a majority of write locks on each data object in its write set. The
various lock managers acknowledge immediately whether the lock is set or blocked. If two transactions concurrently issue an update, only one will achieve majority vote and will be able to proceed.

3.3.1.4 Centralized 2PL

*Redundant copies of objects*

In this technique, one centralized lock manager is present in the system. Before accessing data at any site appropriate locks must be obtained from the central lock manager. Hence both *rw* and *ww* conflicts are detected by the lock manager.

3.3.1.5 Granularity of Locking

Locking may be performed at different granularities. Segments, objects or object fields may be locked depending on the desired application and concurrency. Decreasing the lock granularity improves concurrency. This, however, requires a larger lock table to be maintained and kept up-to-date among the various sites.

3.3.1.6 Deadlock Detection

The preceding implementations of 2PL force transactions to wait for unavailable locks. If this waiting is uncontrolled a deadlock situation may arise (see Figure 3-2).

```
<table>
<thead>
<tr>
<th>T1</th>
<th>Time</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(x)</td>
<td>t1</td>
<td>lock(y)</td>
</tr>
<tr>
<td>request(y)</td>
<td>t2</td>
<td>request(x)</td>
</tr>
<tr>
<td>t3</td>
<td>t4</td>
<td>Deadlock</td>
</tr>
</tbody>
</table>
```

Figure 3-2: Deadlock Situation

The basic characteristic of a deadlock is the existence of a set of transactions such that each transaction is waiting for a resource locked by another transaction that is directly or
indirectly waiting for the first. This situation can be conveniently represented with a wait-for graph. A wait-for graph is a directed graph having transactions as nodes, an edge from transaction \( T_1 \) to transaction \( T_2 \) represents the fact that \( T_1 \) is waiting for a resource locked by \( T_2 \). The existence of a deadlock situation corresponds to the existence of a cycle in the wait-for graph. Figure 3-3 shows the wait-for graph for the deadlock example in Figure 3-2.

![Figure 3-3: Wait-for Graph](image)

Two general techniques are available for deadlock resolution:

- **Deadlock Prevention.** This is a cautious scheme in which a transaction is aborted and restarted when the system detects that deadlock might occur. The basic approach is to assign priorities (possibly timestamps) to transactions and to decide whether \( T_i \) (requesting lock) can wait for \( T_j \) (already owns lock). If the test fails, one of the two are aborted and restarted. For example, \( T_i \) could wait for \( T_j \) if \( T_i \) has lower priority than \( T_j \). This prevents deadlocks because, for every edge \( <T_i, T_j> \) in the wait-for graph, \( T_i \) has lower priority than \( T_j \). Since a cycle is a path from a node to itself and since \( T_i \) can not have lower priority than itself, no cycle can exist.

- **Another scheme is preordering of resources.** This is a deadlock avoidance technique that avoids restarts altogether. This technique requires the predeclaration of locks (each transaction obtains all its locks before execution). Data items are numbered and each transaction requests locks one at a time in numeric order. The priority of a transaction is the number of the highest lock it
owns. Since a transaction can only wait for transactions with higher priority, no deadlocks can occur.

- **Deadlock Detection.** In this scheme, transactions wait for each other in an uncontrolled manner and are only aborted if a deadlock actually occurs. Deadlocks are detected by constructing wait-for graphs, searching for cycles and picking a transaction in the cycle to abort. This is complicated in a distributed environment because transactions may be executing at different sites. Moreover, transactions may be divided into subtransactions that need to be performed at sites where data is stored.

### 3.3.2 Synchronization Techniques Based on Timestamp Ordering

Timestamp ordering is a technique whereby a serialization order is selected a priori and transaction execution is forced to obey this order. Each transaction has a starting time and each piece of data has the timestamp of the transaction that last read it and the transaction that last wrote it. These techniques allow a transaction to read or write a data object x only if x had last been written by an older transaction; otherwise it rejects the operation and restarts the transaction. Generating timestamps in a distributed system is more complex than in a centralized system because of the lack of a global clock. Many techniques have been proposed to ensure the uniqueness of the timestamp [Lamport 78].

#### 3.3.2.1 Basic Timestamp Ordering

**One copy of Object**

The basic timestamp mechanism applies the following rules.

1. Each transaction receives a timestamp TS when it is initiated at the site of origin.

2. For each data object x, the largest timestamp of a read operation and the largest timestamp of a write operation are recorded - indicated by RTM(x) and WTM(x).
3. For reads, if TS < WTM(x) the transaction is trying to read data that was written by a "later" transaction. The read operation is rejected and the issuing transaction is restarted with a new higher timestamp; otherwise the read is executed and RTM(x) is set to max(RTM(x), TS).

4. For writes, if TS < RTM(x) or TS < WTM(x), the write operation is rejected and the issuing transaction is restarted with a new higher timestamp; otherwise the write is executed and WTM(x) is set to TS.

An optimization can be performed to the fourth rule using the Thomas Write Rule (TWR). This states that instead of rejecting a write when TS < WTM(x) we can simply ignore it. Hence, writes that try to place obsolete values are ignored.

3.3.2.2 Multiversion Timestamp Ordering

This technique is based on keeping around old versions of the data object x and results in an improvement of RW synchronization over the basic T/O [Reed 78]. Associated with each data object x is a set of R-ts and a set of <W-ts, value> (version) pairs. R-ts record the timestamps of read operations and the versions record timestamps and values of executed write operations. RW synchronization is accomplished by applying the following rules:

1. Let TS be the timestamp of a read operation on data object x.
   The read operation is processed by reading the version of x with the largest timestamp < TS and adding TS to x's set of R-ts. A read is never rejected.

2. Let TS be the timestamp of a write operation on data object x.
   Let interval(W) be the interval from TS to the smallest W-ts(x) > TS, if any R-ts(x) lies in the interval(W), the operation is rejected, otherwise a new version of x, <TS, new value>, is created.

A read ignores all versions with timestamps greater than that of the read, hence the value read is identical to the value it would have read had it been processed in timestamp order. A write is processed by creating a new version of the object unless a read with a greater timestamp has occurred. In this case the write is rejected.
3.3.2.3 Conservative Timestamp Ordering

This is a technique for eliminating the possibility of conflicting operations and hence eliminating the need to restart transactions because of such conflicts [Bernstein 80]. This approach however provides less concurrency than the basic T/O.

The fundamental idea underlying conservative timestamping is simple: No operation is ever performed until it can be guaranteed that it can not possibly cause a conflict (and therefore a restart) at any time in the future. In other words, a request for an operation from transaction T is delayed until the system knows that no conflicting requests are received from older transactions. Conflicts are eliminated by serializing all operations at each site, not just those operations that would otherwise conflict.

This situation may be improved considerably by introducing the concept of transaction classes. Transaction classes are defined by a read set and a write set. In this case the operations are delayed until no more operations in the readset or writeset with a smaller timestamp need to be performed at various sites.

3.3.3 Synchronization Techniques Based on Optimistic Control

The basic idea of optimistic methods is the following: Instead of suspending or rejecting conflicting operations, like in 2PL or T/O, always execute a transaction to completion. However, write operations are performed only in a local workspace. Only if the validation test is passed at transaction commit time are the write applied to the database. If the validation test fails the temporary writes are ignored and the transaction is restarted.

3.3.3.1 Majority Consensus

This algorithm [Thomas 79] assumes that a copy of each data object is stored at each site. Hence each transaction is executed completely at the site of its origin. Another assumption of this algorithm is that there are no data objects that are written by a transaction but not read. Each transaction receives a unique timestamp when it starts execution, and each
representative of each data object carries the timestamp of the last transaction which has written into it. Transactions execute in two phases. In the first phase an update list for the transaction is produced at the home site. It contains the data objects in the transaction's read-set with their timestamps, the new values of the data objects in the transaction's write-set, and the timestamp of the transaction itself.

In the second phase, the update list is sent to every site to validate and vote on. The voting rule followed by each site is shown below. If the originating site gets back a majority of positive votes, a message is sent to all sites to commit the updates to the database. Updates are only applied if the timestamp of the request is less than the timestamp of the data object, otherwise the update is ignored (this is the TWR mentioned earlier). If a REJ is received, the transaction is rejected. All sites are informed and hence the update list for that transaction is disregarded. A PASS vote prevents deadlocks. A site votes PASS only when the request's base variables conflict with those of another pending request; in order to inform other sites of a potential deadlock situation. The request in question can continue to be considered until sufficient PASS votes accumulate to prevent majority consensus.

Voting Rule

1. Compare the timestamps of the request read-set with the corresponding timestamps in the local database copy.
2. Vote REJ if any value in the read-set is obsolete.
3. Vote OK and mark the request as pending if each variable in the read-set is current and the request does not conflict with any pending requests.
4. Vote PASS if each variable in the read-set is current but the request conflicts (has common read-set variables) with a pending request of higher priority (earlier timestamp).
5. Otherwise (concurrent conflict, lower priority), defer voting and remember the request for later reconsideration.
3.3.3.2 Distributed Certification

One copy of the data

This is a distributed time-based algorithm which operates by exchanging certification information during the commit protocol (Sinha 85). Unlike the previous algorithm it allows distributed transactions (a master transaction at the initiating site and subtransactions at different sites). A read timestamp and a write timestamp is maintained for each data object. Transactions may read and update data items freely, storing any updates into a local workspace until commit time. For each read, the transaction must remember the write stamp associated with the item when it was read. When all subtransactions are completed and have reported back to the master, the transaction is assigned a globally unique timestamp. This timestamp is sent to the subtransactions site in the "prepare to commit" message, and it is used to locally certify all its read and writes as follows: A read request is certified if the version that was read is still the current version and no write with a newer timestamp has been locally certified. A write request is certified if no later reads have been certified and subsequently committed and no later reads have been locally certified already.

Many copies of the data

This requires that sites with representatives of the data also participate with the certification. Updaters simply certify the set of writes they receive at commit time.

3.3.4 Comparison of the Three Methodologies

The locking approach may easily result in deadlock requiring deadlock resolution or detection. The timestamp approach is deadlock free but may require transaction restart, another expensive process. The optimistic approach is based on the assumption that conflicts are rare and therefore most transactions are validated (few rollbacks).

Performance analysis of the 2PL, Basic T/O and distributed certification was conducted by
in the context of distributed database systems. The following conclusions were reached:

- 2PL and distributed certification dominated Basic T/O in terms of performance.
- When the cost of sending and receiving messages was low, 2PL was the superior performer due to its avoidance of transaction restarts.
- When the message cost was high and data was replicated, distributed certification outperformed 2PL due to its ability to exchange the necessary synchronization information using only the messages of the two-phase commit.

3.3.5 Other Concurrency Control Schemes

Many systems (Neptune, Gypsy) that are characterized by long transactions use version merging to resolve version conflicts that may arise because of concurrent long user-driven transactions. The methods proposed require the user's intervention to merge versions together. These schemes fail to take into account the need for serializability in some situations. Some file systems provide mechanisms for automatic merging of versions. Various strategies have been proposed to resolve merging conflicts:

- Conflict resolution rules
- Data Flow [Reps 85].
- Some choice is made, but a comment is also included indicating the nature of the conflict, or the description of the nature of the conflict is placed in a separate document.
- The user is interactively asked to decide which version to keep.

3.4 Security and Protection

Implementation of protection mechanisms that permit sharing fall into two categories [Saltzer 75]:

- Capability Systems
- Access Control List Systems
3.4.1 Capability Systems

A capability is an identifier for a shared object that also grants rights to perform certain operations on the object. Capabilities are typically represented by a bit sequence consisting of a unique identifier for the object and a specification of access rights to the object. A capability may be presented to the system in return for an object. Capabilities must be secure and unforgeable.

The capability system has as its chief virtues its inherent efficiency, simplicity and flexibility. Efficiency comes from the ease of testing the validity of a proposed access: if the accessor can present a capability, the request is valid. The simplicity comes from the natural correspondence between the mechanical properties of capabilities and the semantic properties of addressing.

The potential problems with the capability system are revocation of access, uncontrolled propagation of access and inability to review access. Once a capability has been given to someone it cannot be disabled. It is, however, possible to put time limits on capabilities so that they expire after a given period in time. Capabilities may be distributed to other users without the knowledge of the creator of the object. If the user id is incorporated in the capability then only the users with the appropriate user id can use the capability, assuming that one user can not impersonate another user.

3.4.2 Access Control List System

An access control list is a list maintained for each object in the system indicating the users or domains that have access to the object and the operations permitted for each user or domain.

Access control list systems overcome the problems of a capability system but has
implementations problems of its own. Search through an access control list may be a time-consuming process. Moreover, allocation of space for access control lists, which can change in length, can be a formidable implementation problem.
Chapter 4

Literature Survey

Our literature survey covers a wide range of distributed systems. The systems are divided into three categories: distributed relational databases, object-oriented databases and hypertext systems. Our concern is to explore the approaches to object sharing adopted by other systems, the concurrency control schemes used and the update propagation schemes followed.

4.1 Distributed Relational Database Systems

Distributed-INGRES

Distributed-INGRES [Stonebraker 79] was developed at the University of California at Berkeley, as a distributed version of the relational database system INGRES. It uses 2-phase-locking for concurrency control. Deadlocks are detected and resolved with a centralized deadlock detector. It describes the concurrency control and consistency of multiple copies of data in Distributed-INGRES. The basic approach is primary copy 2PL. Note however, that the commercial product INGRES/STAR as of 1987 did not support Data Replication. RTI has committed publicly to such an enhancement in the future.

R*

R* is a prototype system [Haas 82] developed at the IBM San Jose Research Lab in California. It uses 2-phase-locking for concurrency control. Deadlocks are detected and resolved using a distributed deadlock detection mechanism. When a deadlock is detected, its resolution consists of aborting one of the transactions in the wait-for cycle. R* does not support data replication, however, an alternative to replication was introduced in the notion
of snapshots. Snapshots present a view of the database (possibly remote) which was consistently retrieved at some past time and which is periodically "refreshed" but not updated online.

POREL

POREL [Neuhof 82] is a distributed database system developed at the University of Stuggart. Concurrency control is done by 2-phase-locking, and locks are kept until the end of transactions. In order to avoid deadlocks, locks are ordered: there is a total ordering of sites, and lock requests can not be granted at one site until all higher priority locks are granted. POREL supports data replication and uses a variation of the primary copy approach to propagate updates. Update transactions need to lock exclusively the primary copy and perform updates there. They also create an "intention list" at the primary site that contains the updates to be sent to other sites. At all other sites, copies are marked invalid. Copies are updated by a different transaction which is started either when workload conditions are favorable or when a read transaction must make a "consistent" read.

SDD-1

The SDD-1 prototype [Bernstein 80], developed at the Computer Corporation of America, was the first prototype of a distributed database management system. Concurrency control in SDD-1 is achieved using the conservative timestamp method. This mechanism is not deadlock free and the usefulness of conflict graph analysis in a real-life environment has not been demonstrated. SDD-1 supports data replication and uses distributed update propagation (write all). This means that all copies of the data are updated before the transaction commits.

ORACLE/STAR

ORACLE/STAR [Gref 87] is a commercial distributed database system developed and marketed by Oracle Corporation. It does not support data replication. It uses 2-phase
locking at the record level to resolve write-write conflicts. Read-Write conflicts are allowed, i.e., writers do not block readers. Oracle/Star uses deadlock detection as opposed to deadlock prevention to resolve deadlocks.

Table 4-I summarizes the relevant features of the distributed relational databases above.

<table>
<thead>
<tr>
<th>Distributed DBMS</th>
<th>Replication</th>
<th>Concurrency Control</th>
<th>Update Propagation</th>
<th>Deadlock Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle/Star</td>
<td>No</td>
<td>2 Phase-Locking</td>
<td>Not Required</td>
<td>Centralized Deadlock Detector</td>
</tr>
<tr>
<td>SDD:1</td>
<td>Yes</td>
<td>Conservative T/O</td>
<td>Distributed Update Propagation</td>
<td>Deadlock Prevented Wait-for Analysis</td>
</tr>
<tr>
<td>ORACLE*</td>
<td>No</td>
<td>2 Phase-Locking (Record Level)</td>
<td>Not Required</td>
<td>Deadlock Detection</td>
</tr>
<tr>
<td>R*</td>
<td>No Snapshots</td>
<td>2 Phase-Locking</td>
<td>Non Refresh</td>
<td>Distributed Deadlock Detection</td>
</tr>
<tr>
<td>POREL</td>
<td>Yes</td>
<td>2 Phase-Locking</td>
<td>Variation of Primary Copy</td>
<td>Deadlock Prevention ordering locks</td>
</tr>
</tbody>
</table>

Table 4-I: Distributed Database Features
4.2 Object-Oriented Database Systems

There are two general approaches to implementing a shared object space (hierarchy): centralized or distributed. Present prototypes fall into one of these general categories. The following is not an exhaustive survey of present prototypes; however it gives a flavor for the variety of prototype systems being implemented.

GEMSTONE

Servio Logic’s GemStone [Purdy 87, Maier 87] adopted the centralized approach to object sharing. The GemStone object server connects to various workstations through a local area network. It currently does not allow a database to be distributed among several GemStone servers. Method execution can be performed either by executing messages remotely on the GemStone server or by copying an object’s state to the workstation local cache for manipulation. Objects are represented by deputes that decide whether to forward messages to GemStone or cache the object state and execute the message locally.

Stone provides secondary storage management, concurrency control, authorization, transaction management, recovery and support for associative access of objects. Gem sits atop Stone and adds the capabilities of compiling methods into bytecodes and executing that code, user authentication and session control. Part of the Gem layer is the virtual image: the collection of classes, methods and objects supported by the object-oriented system.

Stone supports multiple concurrent users by providing each user session with a workspace that contains a shadow copy of the object table derived from the most recently committed object table, called the shared table. As objects are changed in a session, the shadow object table adds new nodes that are copies of its shared object table with the proper changes. An optimistic control scheme is used to check access conflicts. When Gemstone receives a
commit message, it notifies all deputies of its intent to commit. Each deputy then flushes any modified cached state to GemStone. Stone checks for read-write and write-write conflicts with transactions that have committed since the time the transaction began. If there are none, the shaded object table of the session is treated as if it were "transparent" on the entries that have not been modified, and is overlaid on the most recent version of the shared table. In this way, the changes made by the committing session are merged with those of other transactions that committed after the committing transaction began. If the transaction fails to commit, the changes in its shadow table are discarded and each deputy is told to invalidate its cached state. The developers are currently exploring the implementation of locking and versioning on top of the shadow scheme.

**IRIS**

Hewlett-Packard Laboratories' Iris [Fishman 37] also chose the centralized approach. It implements the Iris Data Model, which falls in the general category of object-oriented models that support high-level structural abstraction, such as classification, generalization/specialization, and aggregation, as well as behavioral abstractions. However, the Iris Storage Manager is a conventional relational storage subsystem. The capabilities supported by the storage manager include the dynamic creation and deletion of relations, transaction management, concurrency control, logging and recovery, archiving, indexing and buffer management.

The developers are actively exploring extensions to the storage manager to support long transactions common in design applications. A version control mechanism is proposed as the basis for concurrency control in design applications. Users would be allowed to check out versions for extended manipulation. This prevents further access to this object by others. A higher level locking mechanism is proposed to support concurrency control for AI-based applications. This provides a hierarchical lock structure with intention locks, as well as conventional read and write locks.
Distributed Object Server

Textronix Laboratories's [Porter 88] adopted the distributed approach, by implementing a low-level distributed object server in which the global object space is distributed across client workstations and objects migrate to the sites where they are used. This server is intended to form the lowest layer of an object-oriented system that will completely insulate the user from the primitive model of objects provided by the server. The server provides a shared object space of persistent objects. The upper layer of the object-oriented system will augment this model to provide such concepts as typing, message passing and inheritance.

The general architecture of the system consists of a network of workstations. Objects are stored at workstations where they are used. Shared objects are located by communicating to a name server. Objects migrate from one site to another for manipulation. Concurrency control for shared objects is attained via a variant of two-phase locking. A shared object must be obtained before it can be accessed, i.e., the object must migrate from a disk manager (local or remote) into the session’s object server memory. Access to this shared object is blocked until the session relinquishes it. The system provides low level version control in terms of historical objects that are immutable and maintain time of creation. They are explicitly created to capture the state of the object at any moment in time.

Distributed Smalltalk

[Decouchant 86] describes the design of a distributed object manager which allows several Smalltalk-80 systems to share objects over a local area network. Single copies of objects are present. Remote access is done through the use of symbolic links "proxy objects" that contain information about the location of the object. Objects then migrate for local manipulation and associated proxies are updated indicating the object's new site. The design does not address concurrency control to permit simultaneous access to objects.
ObServer

ObServer [Hornick 87] is a persistent object store that is currently used at Brown University as the backend of an object-oriented database system (ENCORE) and as the storage manager for an object-oriented interactive programming environment (GARDEN).

Objects are clustered into segments at ObServer. Object access by a client results in the transfer of the enclosing segment, if not already present at the client workstation. Segments were used to reduce communication traffic in object transfer. ObServer supports an object replication facility, i.e., an object may be placed in more than one segment. Moreover, two clients may have copies of the same segment. The system guarantees automatic update to all copies of an object in the various segments. It also guarantees that a client always accesses the latest committed copy of an object through the use of timestamps. Timestamps are given to transferred segments and used objects in the segment by ObServer. If an object is updated, the changes are sent on to the ObServer. Timestamps are sent to be attached to the copies of the object to indicate the new commit time. If another client tries to access an outdated copy of the object (difference in timestamp of object and segment), the current copy is requested from the server.

ObServer uses a comprehensive locking scheme to provide concurrency control. The scheme allows objects to be locked in a range of locking and notification modes. The various modes permit clients to access (read, write) an object in a restrictive or non-restrictive manner. The non-restrictive mode allows clients to share uncommitted changes made to an object. Communication mode allows a lock holder to be notified of the status of an object, including requests from other clients for that object or a committed update from another client.

\(^2\)One of the basic ideas for improving performance in object-oriented programming is that related objects are clustered together in physical memory.
POSTGRES

[Stonebraker 86, Rowe 86] take the approach of storing a shared object hierarchy in a next-generation relational database management system POSTGRES. POSTGRES's extensions to the RDMS include means of supporting complex objects, of allowing new datatypes and of supporting alerters, triggers and general rule processing. The use of an RDMS has the advantage of eliminating the need to implement an object manager that supports shared access, maintains data integrity and resiliency, controls access to objects and maintains data consistency. As objects are referenced by the application, a run-time system retrieves them from the database. Objects retrieved from the database are stored in an object cache in the application process so that subsequent references to the object will not require another database retrieval. Object updates are propagated to the database and to other processes that have cached the object. Alerters notify the system that an object has been updated.

Table 4-II summarizes the relevant features of object-oriented databases above.

4.3 Multi-user Hypertext Systems

Notecards

Notecards [Trigg 86] is a hypertext-based idea structuring system. The basic object is an electronic notecard containing text, graphics and images. Individual notecards can be connected to other notecards by arbitrarily typed links, forming networks of related cards. Distributed Notecards allows users simultaneous shared access from their workstations to notefiles residing on any machine in the network. Distributed Notecards provides contention resolution at the level of individual cards. The system allows any number of users to simultaneously read and display of a given card. However, permission to make modifications to the card is restricted to one user at a time. All readers of the card are notified when modifications are made. Readers are provided three levels of modification
<table>
<thead>
<tr>
<th>Object-Oriented DBMS</th>
<th>Centralization</th>
<th>Concurrency Control</th>
<th>Versioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMSTONE</td>
<td>Centralized</td>
<td>Optimistic Control</td>
<td>Future</td>
</tr>
<tr>
<td>IRIS</td>
<td>Centralized</td>
<td>Locking</td>
<td>Yes</td>
</tr>
<tr>
<td>TextObject</td>
<td>Distributed</td>
<td>Locking</td>
<td>Low-Level Historical Objects</td>
</tr>
<tr>
<td>Distributed Smalltalk</td>
<td>Distributed</td>
<td>Unknown</td>
<td>No</td>
</tr>
<tr>
<td>OBServer(Encore)</td>
<td>Centralized</td>
<td>Comprehensive</td>
<td>Yes</td>
</tr>
<tr>
<td>POSTGRES</td>
<td>Centralized</td>
<td>Locking</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4-II: Features of Object-Oriented Databases

notices. The reader is notified (1) when someone has declared an intention to modify the card, (2) when the writer actually writes the modified card and (3) when a writer deletes a card. The system provides no special mechanism for dealing with modify collisions at the individual card level; exclusive write permission is simply allocated for an unlimited period on a first-come, first-served basis. Notecards has no support for versions.

Neptune

Neptune [Delisle 86] defines the notion of contexts to support multi-person cooperative efforts. A context is a collection of nodes and links. A mechanism is provided to merge different contexts together to create a new context. Contexts are used to enclose version histories of objects and hence support concurrent access. A context is used as a private
workspace for making updates, the local updates will eventually be merged with another
master context. Conflicts may arise if a modification was made to the master context that
would be obscured if local updates were merged with the master. The system is able to
detect the conflicts but it is up to the user to resolve them.

**Gypsy**

Gypsy [Cohen 88] is a programming support environment built on top of an object-
oriented operating system. The foundation of the system is the Version Manager, which is
used to store, organize, and selectively retrieve versions of objects. Multiple users are
supported by the Workspace Manager. The Workspace manager provides a protected
environment for working on private versions. Objects are clustered into workspaces that
define user's access rights to the objects. Objects are not replicated but various version and
version branches for an object exist. Versions created inside a workspace are private to the
workspace and are not publicly visible until they are released from the workspace.

To create a new private version, a user must first be attached to a workspace, and then must
lock a branch of a version group. Ordinarily only one user can be attached to a given
workspace. However additional workspace access controls permit concurrent access; users
who take advantage of this flexibility have to depend upon synchronization mechanisms
outside those provided by Gypsy (e.g., informal messages indicating who is using what).

Close and weak models of cooperation are provided. Close cooperation allows the creator
of the workspace to specify whether authorized users may access the workspace
concurrently. Weak cooperation allows user to access private versions with specified
access rights. This allows coworkers to get a sneak preview of a needed object before it is
released.

Merging of versions is not presently implemented in Gypsy. The design proposes to
support type-specific merge with parameterized operations that use various strategies to handle conflicts. Possible strategies include conflict resolution rules or interactively querying asking the user.

**Notes**

Even though Notes is not a hypertext system, its support of replication makes it an interesting system to look at. Notes [Greif 88] is a group communication system that is used by people to share textual, numeric and graphical information. The system operates on a local area network of personal computers. Notes is based on document manager that provides permanent storage for free-form and semi-structured objects of arbitrary size. These databases may be replicated at different sites allowing replicas of each document to exist. Notes replication ensures eventual consistency of the documents in all replicas. The database sites communicate to verify that they have the latest copies of the database; if not the newer version is obtained from the remote database.

Optimistic concurrency control is used to detect and signal occurrences of multiple updates to a single version instance of a document (in the same database) and finally converge the versions to a single version of an updated document (by asking the user whether he wants to overwrite an existing version). The system resolves conflicts with different versions of a document at another database site by choosing the version that has been edited and saved most often. Access control lists, may be used to control concurrent updates in a number of ways: (1) to ensure that changes can only be made to a single master copy, and (2) ensure that changes can only be made by authors of the document.

Table 4-III summarizes the relevant features of the distributed hypertext systems above.
<table>
<thead>
<tr>
<th>Hyperext System</th>
<th>Centralized vs. Distributed</th>
<th>Concurrency Control</th>
<th>Versioning</th>
<th>Merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note cards</td>
<td>Distributed</td>
<td>Locking (Write-Write Conflicts)</td>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Neptune</td>
<td>Distributed</td>
<td>Version Control</td>
<td>Yes</td>
<td>Performed by User Difference detection provided</td>
</tr>
<tr>
<td>Gypsy</td>
<td>Centralized</td>
<td>Locking Versions</td>
<td>Yes</td>
<td>Performed by User</td>
</tr>
<tr>
<td>Notes</td>
<td>Distributed Replication</td>
<td>Optimistic Concurrency Control</td>
<td>Versions created internally but not maintained</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Table 4-III: Hyperext Features
Chapter 5
Object Lens Revisited

In this chapter we mainly present a description of Object Lens. For those readers who are familiar with object-oriented database and hypertext systems we will indicate features of these systems that Object Lens shares. We will then present features specific to Object Lens that will play a major role in our choice of the appropriate object sharing scheme for Distributed Object Lens.

5.1 Object Lens as a Hypertext System

Object Lens fits four of the six features of an idealized hypertext system enumerated in [Conklin 87]. The features present in Object Lens are:

- "Windows on the screen correspond to nodes in the database on a one-to-one basis". The nodes in Object Lens are the objects and objects have display windows.

- "Windows can contain any number of link icons which represent pointers to other nodes in the database". Object display windows contain a number of fields and values in these fields. The values can be free text combined with link icons. Clicking on the link icon causes the system to find the referenced object and open an object display window for it on the screen.

- "The user can easily create new nodes and links". Object types and instances are easy to create through a template-based user interface.

- "The database may be browsed". The user in Object Lens may navigate the object space by following links and opening windows successively to view the
content A user can query the system for objects with particular attribute values

5.2 Object Lens as an Object-Oriented Database System

Object Lens provides an interface to an object-oriented database by enjoying the characteristics of object-oriented systems [Bobrow 86]. These characteristics are:

- each object includes a collection of fields and field values, as well as embedded objects.
- each object has a set of actions that can be performed on it by sending messages between objects.
- objects are arranged in a hierarchy of increasingly specialized type with each type inheriting fields, actions and other properties from its parents.

Moreover, Object Lens provides a simple way to perform database queries: users can create agents that scan the objects in one folder and insert links to selected objects into another folder. The rules in the agents specify the criteria for selecting objects.

5.3 Object Lens Specific Features

In addition to the object-oriented features and hypertext features of Object Lens, the following specific characteristics influence our choice of an Object Sharing Scheme for Distributed Object Lens.

1. Naive Users. Object Lens is geared toward non-programming users. This makes user-interface a major component of the system. The user-interface model influences what aspects of object sharing are exposed to the user.

2. Agents. Object Lens supports the creation of rule-based "agents" that
process information automatically on behalf of the user. These agents can be triggered by events such as the modification of a particular object.

3. Semistructured Objects Each object is a collection of fields and field values, however the user may fill in as much or as little information in the different fields as they desire. Fields are not typed and the values can range from free text to a link to another object to a combination of both.

4. Customizable Folders Objects may be collected in customizable folders. These folders can be maintained manually or by an agent. Queries are executed on the objects in the folder in the same manner as queries are executed on tables in relational databases.

5. Version Maintenance. As a tool for implementing cooperative work applications, Object Lens should support the Maintenance of different versions of objects.

5.4 Using Object Lens

Object Lens, like its predecessor Information Lens [Grant 87], makes extensive use of information exchange (literal text) using specialized semistructured electronic mail messages.

Creating semistructured objects requires defining a collection of fields and field contents. The fields are defined by creating a new object type in the type hierarchy. Note that, the new object will inherit fields from its parent in the hierarchy. The inherited fields can be renamed or their display suppressed. Additional fields may be added.

Once the object type is defined (for example PERSON), object instances can be created by filling in the fields in a displayed object window corresponding to object type (e.g., PERSON), hence creating, for example, JOHN and TIM. The contents of an object’s filed
may be a link to another object. This link may be expanded to appear as an embedded object in the enclosing object. Object instances may be modified by editing the window display corresponding to the object. This interactive editing of objects may take on the order of minutes to hours. Actions may be invoked on the object from the command bar shown across the top of the displayed object window. There are several kinds of actions, including actions that apply to the editing window such as close, move and shape, and actions that apply to the object such as save or add link. These actions take on the order of subseconds to execute. Figure 5-1 depicts a displayed object form of a Person object in Object Lens.

Users typically do not have more than about ten objects displayed at one time. This is basically due to limited screen real estate and the incapability of a user to work on many tasks at the same time.

Folders can be created and objects can be added to the folder either automatically by an agent or manually by inserting a link from the folder to the object. Folders can be displayed as tables or trees. Tables show the values of selected fields from the objects contained in the folder. Trees show the links between different objects.
Agents can be created to automatically process information on behalf of the user. An agent, when triggered, applies a set of rules to objects in some folder. Rules may follow links in objects to access information in embedded objects. An agent may be triggered by events such as the addition of links, arrival of new mail, passage of the hour or perhaps modification of a certain field in an object (not yet implemented). A rule contains two parts, the description and the action. The description specifies the pattern that should be matched for the rule to fire (by indicating the values of some fields of the object instances in question). The action specifies the action that should be taken if the rule fires, such as copy, move, delete or add keyword.

A user may navigate the object space by starting at the basic folders, such as the folder containing all folders in the system.

The type of operations that are of interest to us are:

• A user opening an object form, editing the contents of the various fields and then saving the changes.

• A user adding or deleting a link from a folder to an object.

• Automatic execution of rules by agents on objects in the folder.

The latter two take on the order of seconds, the former on the order of minutes or even hours.
5.5 Object Lens Architecture

Object Lens contains the modules [Berenson 89] depicted in Figure 5-2.

![Diagram of Object Lens Architecture](image)

Figure 5-2: Architecture of Object Lens

The Object Manager is currently built on top of the Xerox object-oriented programming environment, LOOPS. LOOPS keeps track of all classes and class-instances and their links to each other. It also keeps track of the current state of each object and execute the methods.

The Window Manager is responsible for presenting the object in window form. It handles user-events (through the window display command bar). Those that deal with display and presentation it handles internally and those that deal with object instances it passes on to the Object Manager. Garbage collection is implicitly performed by the underlying system. An object is garbage collected when there are no more references or links to the object. This process is much more complex in a distributed system.

The Agent Manager is responsible for executing agents. It maintains a clock for triggering
time-driven events and receives messages from the Object Manager about new links, updated objects and manually triggered agents.

Objects are currently saved permanently only in files on a file server. The files need to be loaded for the objects to be defined. The files may remain on the workstation and hence define the Object Lens object space. The objects on the workstation will not survive system crashes (they are not persistent).

Object Lens is presently a single-user system. The set of objects that a user knows about are those on his workstation. Object sharing is limited to sharing the information content of the object. This may be achieved in two ways:

- Mailing literal objects in an Object Lens message
- Loading a specified file created by the exporter of the object at a common file server (this may be done somewhat transparently).

The first approach is in a sense a one-time exchange of object information. Any future changes made to the object by the sender are not reflected in the other object.

In the second approach the user is reloading the state of an object every time it is modified. Propagation update is done through a central file server. This approach also allows for the previous version to be maintained through a link from the current new version. This implementation is slow and is mainly addresses update propagation, only one of the many features required for a comprehensive object sharing scheme.

5.6 An example

A typical application in Object Lens is processing incoming mail into folders according to attributes of the message such as the subject or the sender. The incoming messages are processed by an agent and put into a shared folder that is accessible by a number of users.
Consider the following example. An agent: BUG-MESSAGE-COLLECTOR filters all incoming messages that are pertaining to bug fixes and inserts them into the shared folder BUG-MESSAGES. Any user with access to the folder could browse through and read the messages. Now consider the case where the BUG-MESSAGE folder is opened at a user’s workstation. Any updates made to the folder by the agent BUG-MESSAGE-COLLECTOR because of incoming messages appear in the opened form. Note that earlier we suggested for objects such as a PERSON that the user should be prompted to refresh the object form. We feel that it would be more appropriate in the case of folders, for links to be added without the need for user intervention since folders are in most cases background processes.

In the next chapter we discuss the requirements and features of object sharing for the Object Lens system.
Chapter 6

Multi-user Object Lens

Distributed Object Lens will allow users to simultaneously share access from their workstations to objects residing on any machine attached to the local area network. Objects may be replicated at various sites to improve performance and availability.

Distributed Multi-User Lens will have the following basic features:

- **A distributed object space**: Objects may be located at different sites in the network and may in some cases be transparently accessible by remote Object Lens users. Hence, objects may have links to other objects located elsewhere in the network. The system must support such links.

- **Concurrent users**: In a multi-user system, one or more user may concurrently access the same object. This may lead to loss of information and modifications. The system should provide a mechanism for guarding against such anomalies.

- **Protection of shared Objects**: Access to objects vary from one user to another depending on the authorization of the user. One user may be allowed to only read the object while another is allowed to read and modify the object.

We will first explain some early design decisions made for Distributed Object Lens. Next we define the terminology we use. We then present a design for object-sharing in Distributed Object Lens with emphasis on the following features:

- Shared object creation.
- Shared object deletion.
- Shared object protection.
• Shared object modification that accommodates concurrent users

6.1 Early Design Decisions

The goal of the first prototype of Distributed Object Lens is to test the proposed schemes for object sharing. The choice of the architecture at this stage is of secondary importance. We thus assume an architecture that requires the least effort to implement and that most resembles the current configuration. The configuration is that of the server and the client being on the same workstation. We rely on the local disk manager of the underlying system to act as a primitive object server. This avoids the need to implement an object server or acquire one that fits the specifications mentioned in chapter 2. We need to build the client object manager to provide the features of object sharing. Many of the duties that were assigned to the object server in chapter 2 will now be performed by our designed object manager.

We allow replication to provide fast local access to objects when desired. Objects may thus be located at a number of workstations. Since replication is supported there is no need to support migration of objects. There is also no need for caching, since objects are located at the local workstation, not at a remote object server.

The objects at the workstation are not persistent (in case of a crash the objects need to be reloaded). Future prototypes will need to incorporate persistent object stores. This would be easy to add in this architecture.
6.2 Terminology

*Conceptual Object*: is an object that directly represents a real life entity such as the person John Smith with phone number 547-0741 and address 100 Memorial Dr. There is a one-to-one relationship between a conceptual object and the real life counterparts.

*Object Instance*: This refers to object entities in the object-oriented language. These will have Unique Object IDs. However, object-oriented languages do not guard against the occurrence of more than one object instance for a conceptual object. Hence, if an object is duplicated, a new object instance is created.

*Object Instance Representative*: This refers to distributing the object at different sites in the network. The different object instance representatives should be treated as the same object instance. This is important if we are to allow different copies of the object instance to physically exist at different workstations. The association between the different representatives is essential to ensure update propagation to all representatives that exist. Object UTD may be used to associate each object instance with its various representatives.

*Object Instance Version*: This refers to a version of the object instance mentioned above. This permits revisions to be made to the object, with possibly a history list of all outdated versions. This is not implemented in current object-oriented languages, but may be implemented by generating object UTDs to associate the different versions with the same object instance.

*Object Space*: A user's object space consists of all objects that he can access through a series of links directly or indirectly from the initial node. The nodes and the links are arranged together in a network structure.
6.3 Objects in Distributed Object Lens

6.3.1 Shared vs. Personal Objects

An object in distributed Object Lens is either a personal object or a shared. Shared objects may be public or private.

**Personal Object**

Associated with each user in the Object Lens network is a set of personal objects. The user is solely responsible for modifications made to the objects and is solely guaranteed to see those modifications. These are equivalent to the user's objects in the current single-user Object-Lens system.

However, a user may create a new object instance of a conceptual object and give it to another user in the system. Neither the creator of the object nor the system is responsible for the maintenance of the object instance. The new object instance is the property of the receiver. No restriction is made on where a personal object instance physically resides. It may be local at the owner's workstation, or at a remote site, such as a database.

**Shared Object**

Conceptually a shared object is one to which at least two users have some form of access. Sharers of the object are guaranteed to eventually see the modifications made to the object.
Each Object Lens user will view his personal objects plus a set of shared objects. He will have full access rights to his personal objects, but a varying range of access rights to the shared objects.

Consider a two-user Distributed Object Lens system as shown in Figure 6-2. User's A object space consists of the objects PROCESS, LENSFOLDER, TOM, JOHN, KEVIN, TASKX, TASKMANAGER, and acct T. User's B object space consists of the objects PROCESS, LENSFOLDER, KEVIN, TIM, JOHN, TASKX, and acct K. A's personal objects are acct T and TASKMANAGER. B's personal objects are acct K and TIM. The remaining objects are shared by both A and B.

Figure 6-2: Two-user Distributed Object Lens

6.3.2 Private vs. Public Objects

The distinction between private and public objects is that of authorization and security. Authorization describes access rights by users to various objects. Access rights fall into a combination of the following [Read, Modify].
Public Objects

Any user in the Distributed Object Lens network has all access rights to Public Objects. Public objects are a subset of shared objects. Public objects are shared by all distributed Object Lens users with full access rights.

Private Objects

A private object may only be accessed by an authorized user. Access may be limited to any of read, modify or delete. Shared objects may be divided into sets of private objects, each set adhering to authorization requirements.

An example of a private object is the LENSFOLDER object in figure 6-2. Both users A and B have access to it because they are part of the LENS group. A non-LENS group member will not have access to the folder. Hence, the LENSFOLDER is private to Lens group members. Moreover, user A might have "delete" rights to the TOM object while user B only has "read" rights to the object.

6.3.3 Local vs. Remote Objects

In Distributed Object Lens objects may be physically located at any workstation in the network. A user may access objects that are local to his workstation or that are remote at another workstation or possibly a database server. It may be necessary to expose this distinction to the user to explain the delay time experienced in accessing the object. This distinction is not as important in a fully redundant implementation of Distributed Object Lens (i.e., each object is replicated at every site in the network).
6.4 Creation of a Shared Object

In this section we will describe user interface for creating shared objects.

**Granting access to a Shared Object.**

The actual object is created in the same manner as objects are currently created in Object Lens (i.e., using the editor to fill in the slots of a semi-structured template corresponding to the object type). Once the private object is created, the owner may decide that other users should also have some form of access to the object. Informing or granting access to the object may be done in one of two ways:

1. The creator sends a Lens message to another user, with a link to the object. The receiver then inserts a link to that object into another object (e.g., a folder) in his object space. The receiver can then, or at a future time resolve the link, allowing him to view the contents of the object.

2. The creator inserts a link to the object in an object that is already shared. The system can possibly notify other users of the addition of this new object. A special case, and possibly the most common, would be the insertion of the link into a "shared folder." The former of the two methods is appropriate for initiating sharing of a parent (root) object. We will illustrate the two methods using the example in Figure 6-2.

User B creates a task object TASKX. User A manages all tasks by using his personal TASKMANAGER object. User A therefore needs to have access to TASKX through a link from the TASKMANAGER object. User B therefore mails a message to user A with a reference to the TASKX object (Method 1). User A can then adds a link from the TASKMANAGER object to TASKX. The reference may be later resolved.
The LESSFOLDER folder is shared by all users (A and B). This implies that all objects that can be reached by tracing links from the folder are shared by all users. If a new user joins the group, a new person object instance (JOHN) is created corresponding to the user, and a link to the object instance is inserted in the LESSFOLDER folder. All users will now share the new object instance. The JOHN person object was made accessible to other users by inserting a link to it from an already shared object LESSFOLDER (Method 2).

**Resolving Links**

Resolving a link could imply one of two things depending on the desired implementation.

- Creating a local object instance representative of the object. The shared object becomes a local object and future accesses will be made to the local copy.

- Remotely retrieving the object from its site. The object remains a remote object and future accesses to it will also result in remotely retrieving it.

**IMPLEMENTATION**

Each object may have a number of object instance representatives that are located at different sites. A unique object identifier should encompass information about the location of the particular object instance representative and the object it corresponds to. A possible configuration for the system wide object id is showed in Figure 6-3.

![Figure 6-3: System Wide Unique Object ID](image)

The object id part is used to associate different object instance representative together. The [Machine Created on] field of the representative/version id indicates the location site of the
representatives. The \textit{[Time Created]} fields allows for different versions to exist. We cannot rely on the object id generated by the underlying system to fit our requirements. If the underlying system allows an object id to be specified, then we choose to specify the unique object ids as mentioned above. Another alternative is to build one level of indirection where object ids generated by the system are mapped to the proposed system wide unique object ids.

Resolving an object reference translates into a read request directed to the object manager. If the object is not local, the \textit{[machine created on]} field is used to direct the read request to the appropriate object manager across the network at site \textit{[machine created on]}.

\textit{Shortcomings of Protocol}

In essence a user is granted access to a shared object by providing the user with a link to that object. Thus sharing protocol has some shortcomings:

- \textit{Unintentional Links}. User A might unintentionally provide links to objects he regards as personal. For example, A grants B permission to include object "x" is his workspace, however, in doing so, A grants B permission to unintentionally include A's personal object "y" because of the link between "x" and "y".

- \textit{Propagation of Access}. User A has no control of who else might get access to an object. For example, if A grants B permission to access object "a", B in turn might give C a link to that object, hence allowing C access to object "a" without A's consent.

- \textit{Revocation of Access}. User A cannot revoke access to an object to which he gave out a link.
6.5 Deletion of a Shared Object

In object-oriented systems objects are removed from memory using a process called garbage collection. The effect of deleting an object from a user's object space is removing his links to that object. Only when there are no more links pointing to an object will the object get garbage collected. In a distributed environment, matters are not as simple. Multiple users may have links to an object. Many of these links are from remote sites. Hence, the reference count that maintains the number of local references to an object should be extended to include remote references. The object is garbage collected when both the local and remote references are zero.

This implies rewriting the underlying garbage collector. This is hard to implement and might be unnecessary for the initial prototype. An alternative simpler scheme would be to use the current underlying garbage collector. However, the object manager has to also maintain local representatives (in the form of links) of remote objects pointing to the local shared objects. These links are dropped when the clients indicate that they have dropped these links.

6.6 Protection of a Shared Object

Thus far we have explained how a user may get a link to an object but we have not specified the type of access the link provides. Distributed Object Lens should provide for controlled access to an object. The question is how such information is declared and enforced?

In Chapter 3, section 4 we saw two models of protection, one independent of object naming (Access Control List) and one dependent on object naming (Capability System). We will examine how each of those models would be tailored to the Distributed Object Lens environment.
Capability System

The object LID described earlier would also consist of a specification of access rights to the object (perhaps containing one bit for each class of operations applicable to the object). When a new object is created by a user, the object manager constructs the OUID of the object with a full set of access rights. The modification of access rights may be made part of the object manager duties. The object (i.e., the OUID) is presented to the object manager with a specification of the rights required. The new OUID returned would have a diminished set of access rights (otherwise users with few rights could apply for more). This new OUID could be mailed to different users that may now have restrictive access to the object.

This scheme as it stands assumes that users do not forge links or impersonate another user. Users would do this to increase their access rights or to get access to objects that they are not authorized to get access to. For our purposes of designing a research system assume no malicious intent on the part of the users. An authentication mechanism such as Kerberos could cleanly be added to a production system to guard against possible forgery and impersonation.

Access Control List:

A list is associated with each object specifying the authorized users and the operations permitted for execution by the users. Such information is separate from the OUID and is maintained by the object manager. Hence, when a user requests access to an object in some specified mode, the object manager verifies that the user is allowed such access before returning the object. From the user's point of view, when creating the object he needs to specify the users that have access to that object and their access rights. Another option would be to specify a set of rules that govern access to the object. These may be easier to formulate and have the effect of defining protection domains for the object. This however, may be inefficient because it requires resolving the rule every time the object is requested.
Comparison of Capability-Based and Access Control List Systems

A capability-based protection model does not guard against the shortcomings mentioned in the earlier section. An access control-based protection model incurs a large overhead that might be unnecessary in a lot of cases. Given the hierarchical structure of the object space, the question arises whether a child object inherits the properties (access rights) of the parent object. This may be appropriate in some cases but not all. It is possible to allow this as a default, but explicitly overwrite it by describing a new set of authorization rules for the child object.

Hybrid Model

It is more appropriate to use access control lists in the restrictive sense as opposed to the permissive sense. The access control lists will contain information regarding users that are not allowed access to the object. A hybrid implementation is the most adequate.

Restrictive access control lists are used to overcome the deficiencies of a capability-based system. It is important to guard the object against undesired users who have somehow obtained a link to the object. Hidden links to personal objects may be protected using an access control list that denies access to all users except the creator of the object. Hidden links may be protected in two different modes:

- The unauthorized user may have knowledge of the existence of a certain object, and hence can see it in its icon form. The user however, is not permitted to view the contents of the object. To follow on the example in Figure 6-2, an unauthorized user may be permitted to see that Tom is a member of the Lens group, but is not permitted to get more information about Tom by viewing the contents of TOM person object.

- The unauthorized user is oblivious to the existence of the object, i.e., the links are hidden. Hence, when the unauthorized user looks at the LENSFOLDER folder he will not see a link to the TOM person object.
The latter mode is much harder to implement. It requires that the object server performs the following additional steps before returning the object:

- Follows and resolves any links the requested object may have.
- Verifies if access is permitted to these embedded objects.
- If so the links are kept intact.
- Otherwise remove the links.

**Shared Agents**

Agents in Object Lens are implemented as objects, and hence can be shared like any other object instance. Consider an agent that operates on the LENSFOLDER folder. Both user A and user B have access to the agent and hence can trigger it on the folder. However, user A is restricted from seeing some objects in the LENSFOLDER, namely TIM. The outcome of the triggered agent should differ from one user to another. Hence, a triggered agent should have the same access rights to the objects as the user that triggered it. A triggered agent is analogous to an executing program. It will have the same userid as the user who triggered it. When the rules are fired and the actions are executed, the objects that are read or scanned are those that can be read by the user.

Another option is to support agents with parameterized userids that may be set by the creator of the agent. The default is for an agent to assume the userid of the user that triggered it. Otherwise, the userid of the agent may be set by the creator to be some userid (e.g., himself). Hence, even if the agent is triggered by another user, it scans the set of objects that are accessible by the specified userid. This is helpful in giving a user access to the services of an agent to run a particular query without giving him access to the entire underlying folder. The user will only have access to the objects in the folder that satisfy the query.
User Interface

In accessing an object, a user views the corresponding window display. Depending on the user's access rights, the appropriate commands are shown on the command bar. For example, an edit command shows only if the user has the right to modify the object, and a delete command shows only if the user has the rights to delete the object.

Specifying restrictive access lists is an extra command when the object is created.

6.7 Modification of a Shared Object

Objects in Object Lens can be modified either interactively by the user or automatically by triggered agents. Object Lens currently supports the following user-driven commands that result in a change in the object space: save, add-link, move, close. Users may edit an object and save the changes or add an item to a folder. Triggered agents apply a set of rules (rules can also be applied manually). When a rule is applied, a collection of objects is scanned to look for objects that match the description of the rule. If a match occurs, the action part of the rule is executed on the object. These actions may vary in complexity (see section 6.7.1.2 for further detail).

In a distributed multi-user environment, consistency and concurrency are of paramount importance. Distributed Object Lens should provide support for each of these features. The remainder of this chapter will be devoted to exploring various ways of providing concurrency control for simultaneous users and achieving consistency between different object instance representatives (if present).

Consistency

The basic premise is that all changes made to the object should be made visible within a given time period to all authorized sharers of the object. The question is whether the update propagation should be exposed to the user or done transparently by the system. This is only
an issue if there are different object instance representative of the object. The two options are:

1. **System Guaranteed.** Once an object is modified, the system guarantees that all sharers of the object eventually see the change (loose consistency control). How this is implemented depends on the shared object implementation we choose and the concurrency control desired.

2. **Exposed to user.** The modifier of the object sends a message indicating the changes to all sharers of the object. It is then up to the sharers to modify their object instance representative

**Concurrency**

The system should allow any number of users to access the same shared object at the same time. It is vital in such a situation to ensure that concurrent action do not interfere with each others operations. Two kinds of conflicts may arise:

- A **Version Conflict** that results when different users are accessing their object instance representatives concurrently, but are modifying different slots or fields in the object. Version conflicts can be resolved by merging the two representatives together.

- A **Serializability Conflict** occurs when two or more concurrent users are allowed to access the same slot in the object and one or more of the accesses is a modify or when two users are performing modifications that are dependent on each other.

A concurrency control scheme should be adopted that will resolve such conflicts. Again the concurrency control can be done by the system invisibly or visible to the user. Two possibilities arise:
1. **System Guaranteed**: The system should guarantee resolving conflicts. Hence, any transactions that result in a conflict should be aborted and restarted.

2. **User Alarmed**: The user is alerted of a conflict as soon as it is known to the system. It is up to the user to undo the actions to resolve the conflict.

### 6.7.1 Transactions in Object Lens

We will use the transaction model to describe the two types of modifications possible in Distributed Object Lens. A transaction is defined as a unit of work consisting of a sequence of operations, beginning with a special `BEGIN_TRANSACTION` operation and ending with either a `COMMIT` operation or a `ROLLBACK` operation. `COMMIT` is used to signal successful termination (the unit of work has been successfully completed). `ROLLBACK` is used to signal unsuccessful termination (the unit of work cannot be successfully completed - a data object can not be read).

The two types of modifications are formally described in terms of transaction as such:

1. **User-driven modifications** are performed by the user editing the object form. We will term these **interactive transactions**. These transactions are long-lived and may last from minutes to days, i.e., the time that elapses between opening the object form, making the changes and finally closing the object form may be days. Moreover, these transactions are very expensive to roll-back and redo. Interactive transactions will overlap more frequently because of the duration of the transactions.

2. **Agent-driven modifications** are performed as a consequence of a fired rule. Rules are either applied by triggered agents or by users. We will term these **automatic transactions**. These transactions are short-lived, i.e., last no longer
than a few seconds. Automatic transactions are comparable to data processing transactions in database systems.

We will initially examine how the various synchronization techniques proposed for distributed database systems, CAD systems and Hypernext systems may be implemented to achieve concurrency control for interactive transactions. Accordingly we will examine how these techniques may be used to achieve concurrency control for automatic transactions.

### 6.7.1.1 Interactive transactions

An interactive transaction could either consist of a simple read operation (opening the object form) or a read operation followed by editing the object form and then closing the object form hence saving the changes. We will assume that in case of modifications, the transition for \( O_i \) to \( O_{i+1} \) is solely based on \( O_i \), that is, no other objects need to be read. Figure 6-4 shows the two possible sets of interactive transactions. Opening an embedded object is considered as a separate (not a nested) transaction. Our claim is that opening an embedded object is no different from opening any other object in the system. The user may feel that the two transactions are related and may require to ensure no interference from other users. We see below how he may accomplish this.

![Figure 6-4: Interactive Transactions](image)

If two users are concurrently accessing an object the following three scenarios are possible:

- Two users are concurrently reading the object -- no conflict arises.
- Two users are concurrently modifying the object -- conflict potentially arises.
• Concurrently one user is reading and the other is modifying the object: a conflict potentially arises.

Since the potential for conflict depends on the nature of the simultaneous interactive transaction, the mode of access needs to be explicitly specified by the user. In many cases, the conflicts that arise are version conflicts, not serializable conflicts. This means that two users are performing independent modifications to the object. The following are three examples that illustrate the three kinds of conflicting conflicts that may arise in interactive transactions.

Consider a TASK object that has as one of its fields [progress report]. The contents of this field is free text that allows users involved in the task to write about their progress. Two users may simultaneously edit the corresponding object form to note their evaluation. Even though both users are accessing the same field, they are installing independent pieces of free text that reflects their evaluations. We refer to this as a version conflict.

Consider a PERSON object JOHN. Two users may simultaneously access the object, one to modify the [phone number] field and the other to modify the [salary] field. We also refer to this as a version conflict.

Consider the TASK object again. Assume that two users are accessing the object simultaneously. However one user’s comments are dependent on the other user’s evaluation. We refer to this as a serializable conflict. Access to the object needs to be performed in serializable order.

User Interface

One option is to let the user specify access when the object form is opened. Hence, a user may access an object either to Read or to Modify. Another option is to have edit as one of
the bar commands on the object form. Opening the object form is an indication of a read and clicking on the edit command is an indication of a write.

6.7.1.2 Automatic Transactions

So far we have seen changes in the object space initiated and explicitly completed by an interactive user through the use of object forms.

Users of Object Lens can also create rule-based agents to automatically process information content of objects. A rule consists of a description and an action. Description specifies when the rule should fire and action specifies the consequence of the rule. The matching of a rule and the execution of its action may be modeled by the transaction: $T_R : R(O) \rightarrow R(O_k) \ldots \rightarrow A(O_1)$ where $A(O_1)$ can either be a simple write $W(O_1)$ in the case of actions of the form (move object to folder, copy object to folder, delete object from folder) (Category 1 rules) or a series of reads $R(O_1), \ldots, R(O_n)$ followed by a write $W(O_1)$ in the case of actions of the form (set <variable> to (calculation)) (Category 2 rules). Figure 6-5 summarizes how rules are modeled as transactions. There would be a number of reads if the predicate (description) part of the rule consists of an embedded object. The write in $T_R$ is dependent on the value of the object read. The consequence part of a rule is conditional upon the value of the predicate. This means that rule execution may lead to serializable conflicts if rules are allowed to interleave. Rules need to be executed atomically to preserve the serial order of the execution. We consider rules and not rule sets to be the unit of work.

Category 1 Rules

Category 1 rules perform their reads on any kind of object but perform their writes specifically on folders by adding or deleting links from folders. The write operation may be viewed as an append operations, hence two different writes may happen in any order without altering the resultant folder. This means that ww conflicts of category 1 rules may be resolved by merging the two modifications together.
$R \land$ conflicts of category 1 rules need to be handled with much more care. Recall that the execution of a write operation of a rule is dependent on the value of the read. Hence interleaving the execution of the two rules such that the read of one occurs before the write of the other may lead to conflicts and inconsistencies. We illustrate this in the following example. Consider the following two rules in pseudo form:

- If object = TOM in Folder A then add link from Folder B to object
- If object = TOM not in Folder B then remove link from Folder A to object

These two rules basically ensure consistency between Folder A and Folder B: object TOM is either in both folders or none of the folders. Figure 6-6 shows how the rules could interleave. The first rule performs $R_1$ and $W_1$ and the second rule perform $R_2$ and $W_2$. Initially the object TOM is in Folder A.

In executions (1) and (4), the rules are performed atomically in serial order. Even though the outcomes are different, they are both consistent with the rules. Hence the order in which the rules are performed atomically is not important as long as they are performed atomically. Executions (2) and (3) depict the wrong outcome if interleaving is allowed. $R \land$ conflicts of this type may be resolved by serializing the execution of the two rules.
Figure 6-6: Interleaving of Rules

Category 2 rules

Let us consider the second category of rules, $T_R: R(O) \ldots [R(O_1 \ldots W(O_n))].$ These procedural actions are more comparable to database transactions and in most cases need to be serialized. To illustrate this consider the following rule in pseudo form:

- If object = task then set taskmanager.count to taskmanager.count + 1

and consider two agents that apply this rule when a new task object is added. If two users concurrently create a new task object, then the above rule is triggered by two different agents concurrently. If the execution of the rules is not serialized one task may be unaccounted for. The net affect would be adding 1 instead of 2 to field taskmanager.count, since both rules will read the initial value of taskmanager.count. Moreover, the actions performed by such rules are not as simple as adding links. Hence both rw and wr conflicts need to be serialized.

6.7.2 Concurrency Control

6.7.2.1 Time-Stamp Ordering

In this section we will describe how timestamp ordering may be used as a synchronization technique in Object Lens. Object access is analogous to the current Object Lens interface ("click on icon " to read, "Close, Save" to write) as opposed to the two modes required if locking is used. The timestamp of the transaction is recorded when the user opens the
object form. Objects will also have the timestamp of the latest read (RTM'O) and the timestamp of the latest write (WTM'O).

As indicated in Chapter 3, Basic T/O is only applicable in an environment where one representative of the object exists. Recall that timestamping is an optimistic approach where transactions are allowed to proceed until a conflict is detected (by comparing timestamps), then one of the transactions is aborted, rolled back and redone. In the case of interactive transactions (R(O) -- W(O)) this implies that conflicts are detected at commit time (when user saves modifications). The write is rejected. A user only finds out that his modifications can not be accepted at commit (save) time. This is highly undesirable in an interactive environment.

Timestamping may be used to serialize rules execution. This however requires that rules be reexecuted whenever conflicts arise. Timestamping does not accommodate for the special treatment of ww conflicts in category 1 rules.

To summarize:

1. Timestamping is inadequate and inconvenient for interactive transactions.
2. Timestamping may be used for serializing category 2 rules. Overhead incurred in maintaining timestamps and reexecuting rules.

Another limitation to basic T/O is its assumption of one copy of an object. Distributed Object Lens would mostly likely support multiple copies of an object.

6.7.2.2 Optimistic Concurrency

Recall that the basic idea of optimistic concurrency control is to always execute a transaction, then validate it and commit it if the validation test is satisfied. In Distributed Certification, a write to an object is rejected if any later read to the object is validated at any other site that contains a representative of the object.

In case of interactive transactions, this means that writes are rejected at commit time if a
simultaneous read or write occurs. This again is inconvenient in an interactive environment. Distributed certification may resolve \( rw \) and \( rw \) conflicts of category 2 rules. It may also be used to resolve \( rw \) conflicts but is unable to treat \( wv \) conflicts of category 1 rules as append-only transactions.

This scheme may be improved if the granularity of update is reduced from an object to a slot in the object.

**Difference between Basic T/O and Distributed Certification**

Given our simple interactive transactions (Reads or Modifies), Time stamping and Optimistic Control are very similar. The optimistic control described above under these conditions is the many copy variation to Basic T/O. This observation is not true in database systems where transactions consist of multiple reads and writes of differing objects.

From the user point of view both approaches look the same. However, since optimistic control makes a working copy of the object, one in principle can get access to that copy and only make the necessary changes to resolve any conflicts. This idea suggests versions as a synchronization technique.

**6.7.2.3 Two-Phase Locking**

Two-Phase Locking was described earlier in chapter 3 in the context of distributed database systems. Several techniques (primary copy 2PL, centralized 2PL) were proposed for an environment where several representatives of an object exist. Two-phase locking resolves \( rw \) and \( wv \) conflicts by blocking read and write operations if a write is occurring, or by blocking a write if a read is occurring.

In the context of interactive transactions this means that if a user is modifying an object no other user may be able to modify it or read it. Recall that interactive transactions may last a
long period of time (e.g., a couple of hours). This means that other users would be blocked from seeing or modifying an object for such long periods of time. This is unacceptable in cases where the modifications to an object are independent. It is however necessary in cases of dependent modifications. Locking works well for serializable conflicts but imposes unnecessary strictness in the case of independent conflicts. Two-phase locking typically block readers if a modify operation is being performed. This again is too strict in the context of interactive operations. We would prefer that readers be allowed to read an object even if it is not consistent with the modifications currently being made to the object.

Serializability is not required for category 1 rules conflicts. Hence locking would be unnecessary. Locking, however, would be appropriate to resolve conflicts of category 1 rules and serialize the execution of category 2 rules.

To summarize:

1. Locking is too restrictive for most interactive transactions.
2. Locking is unnecessary for many executions of category 1 rules. Writes do not need to block other writes because of the append nature of the writes.
3. Locking is necessary to serialize execution of category 2 rules.

Discussion and Issues

Similarity of Different Locking Implementations

In all cases the Lens user has to wait if another user is accessing the object in a conflicting manner. The user is notified once the lock is released. The disadvantage of locking is disallowing access to an object for lengthy periods of time.

Differences

The difference arises in the improved access time (because of local access) and increased availability in the case where "many copies of the object" exist. Replication, however, results in a greater message overhead.
Deadlock Free

With our model of interactive transactions there is no risk of deadlock. This is because interactive transactions are simple (i.e., do not involve reading or modifying more than one object). This is not true for automatic transactions.

Granularity of Locking

The granularity of the locks affect the degree of possible concurrency. Locking at the slot level will result in more concurrency than locking at the object level. Objects in Object Lens may be hierarchical in structure, i.e., a student object may contain a link to a department object. This raises the question whether locking an object means locking all the objects that it has links to. If so, this will result in less concurrency. We indicated earlier that we chose to model the viewing of embedded objects as simple non-nested transactions. This eliminates the need for hierarchical locks. Locking an object will not lock all its embedded objects. Embedded objects are locked when they are accessed. A more appropriate solution would be to maintain the granularity of the locking at the object level. If a user tries to access a link from an object, he may do so in two modes (Read & Modify). The request is granted if no conflicting locks already exist on the object.

6.7.2.4 Version Control

This approach (used in CAD systems) resolves conflicts by merging various versions of representatives together. No waits or restarts are necessary. Version control is an extension of optimistic concurrency control where the working copies are made accessible as new versions and validation and verification are performed by the user through merging. No system known to date is capable of providing automatic merging. Let us consider two users A and B concurrently accessing object "x" in modify mode. Two new versions of "x", "x.A" and "x.B" are created. At this point each version excludes the other users changes. The two versions need to be merged into one that would be the current version of "x". The question is how, when and who merges the two versions together. The possibilities are:
• The two versions need to be merged when a user attempts to read 'a' and sees more than one current version present.

• The two versions are merged by the last user to commit.

The disadvantage of version control is that it provides no mechanism to resolve serializable conflicts. This occurs when changes made by one user are dependent on changes made by another user working concurrently. Notice that in this approach the distinction between object instance representative and object instance version becomes vague. The distinction is made explicit if a long version history of an object is going to be maintained.

6.7.2.5 Proposed Synchronization Technique: Hybrid of Locking and Versioning

So far we have eliminated timestamping and optimistic concurrency control as suitable synchronization techniques for interactive transactions in Distributed Object Lens. Locking is suitable to avoid serializable conflicts but is too harsh and inconvenient for version conflicts. Version control has the opposite problem. It is suitable for version conflicts but does not guard against serializable conflicts. We propose that the synchronization technique best suitable for Distributed Object Lens is one that combines both version control and locking.

Our hybrid synchronization technique uses version control to resolve version conflicts and exploits explicit locking to resolve serializable conflicts in the case of dependent modifications in interactive transactions, category 2 rule execution and rw conflicts of category 1 rules. The locking scheme described earlier used implicit locking enforced by the system whenever a read or write is performed by a user or a rule transaction. Explicit locking allows the user to lock an object, so that crucial modifications to be made to the object with no interference from other users. Another user trying to concurrently modify the same object would be informed of its status and would have to wait for the lock to be
released. Readers however will not be blocked and will be allowed to read the latest committed version with an indication that it is being modified.

Interactive Transactions

A user is also provided with the option of modifying the object without locking it. A laissez-faire attitude (version control) is adopted here and concurrent users are allowed to modify an object in their own workspace and create a new version of the object. At commit time the new versions are broadcast to all sites that have a representative of the object. A user is notified of the incoming version (e.g. by shading the object form). The concurrent user may then decide that irrespective of other changes made to the object he still wants to make his modifications. The incoming version is ignored and user’s version becomes the latest version. Only committed modifications are seen. The user is made aware of uncommitted concurrent modifications. The user may also view earlier versions of the object. Our adaptation of version control relies on the user to resolve version conflicts; the system just detects potential version conflicts and notifies the user of their existence.

Modify-Modify Conflict

At time t user A opens the form of object “x” in modify mode. All sites carrying representatives of “x” are notified. User B opens the form of object “x” in modify mode at t+1. He is notified that “x” is being updated but decides to go on. All sites carrying representatives of “x” are notified that B is also modifying object “x”. User A is then made aware of the existence of another modifier. At this point both A & B know about each others’ existence. It is the responsibility of the user that commits his changes last to merge the other user’s work (assuming both are authorized users). B commits his changes at time t+2. The new version $V_{t+2}$ is sent to all sites as the most current version of object “x”. A is notified of the arrival of the new version. A might choose to ignore this version or compare
changes in $V_{t-2}$ to his changes and merge the two appropriately. In either case user A commits her changes at $t-3$ creating a new version $V_{t-3}$. This is now the latest version of object $x$. It is broadcasted to all sites that contain representatives of "$x" and installed as the latest version.

![Diagram]

Figure 6-7. Resolving Modify-Modify Conflict using Hybrid Implementation

**Read-Modify Conflict**

Referring to Figure 6-7, any reads that occur before time $t$ will result in version $V_t$ being read. Reads between time $t$ and $t-2$ will also result in $V_t$ being read but with an indication that object "$x" is being modified. A user reading object "$x" between times $t-2$ and $t-3$ reads $V_{t-2}$. A user reading object "$x" at time $t-4$ reads version $V_{t-3}$.

So far we have not addressed what happens if a modify is done during a read. Consider the overlap depicted in Figure 6-8.

User A reads object "$x" at time $t$. Version $V_t$ appears in the object form. At time $t-1$ user B begins to modify object "$x". User A is notified of the fact that another user is modifying "$x". User A may choose to terminate his read until the new version becomes available or to continue the read. Once the modify is committed at $t+2$, user A is prompted to refresh his view of object "$x".
Automatic Transactions

Rule transactions need to acquire the appropriate read and modify locks before performing the reads and writes. In interactive transactions we had no need for read locks because users are always allowed to read the most recent committed version of an object.

Consider first category 2 rules. Locking ensures that the rules are performed atomically in serializable order. No two rule transactions may hold conflicting locks on the same object. Conflicting locks are modify-modify and read-modify.

Category 1 rules are treated slightly differently. Only read-modify locks are considered conflicting. Two category 1 rules would be allowed to simultaneously perform their append-only write. The resulting versions may simply be merged.

Deadlock

In the case of interactive transactions we disregarded deadlock because the transactions were simple (i.e., only could write the single object that was read). We cannot make such an assumption in the case of automatic transactions. Consider the rule

- If object = task then set taskmanager.count to taskmanager.count + 1

and two agents concurrently applying this rule. The first agent T1, reads object O and then reads and writes the taskmanager object O1. The second agent T2, reads object O2 and then
read and write the taskmanager object \( O_1 \). Figure 6-9 illustrates a possible deadlock that may arise:

\[
\begin{align*}
T_1 &: R(O_1) \ldots R(O_2) W(O_1) \\
T_2 &: R(O_1) \ldots R(O_2) W(O_1)
\end{align*}
\]

```
T1 - Readlock(O_0)  T2 - Readlock(O_2)
Granted
Granted
T1 - Readlock(O_1)  T2 - Readlock(O_0)
Granted
Granted
T1 - WriteLock(O_1) T2 - WriteLock(O_1)
 Denied
 Denied
```

Deadlock

Figure 6-9: Rule Execution Deadlock

We could choose to prevent deadlocks is such cases by insisting that rules obtain all locks in an ordered manner. Lock ordering avoids deadlocks and enforces a serial order execution.

Another possibility is to detect deadlocks using timeouts. If execution of rules have been halted for a specific period, this is an indication of a deadlock. If a deadlock is detected, rules execution need to be rolled back and reexecuted.

In executing rules the latest committed version should be read unless it has been locked (either by another rule or a user). The system should permit the user in case of an overlap between interactive read and a rule write to view the changes made by the rules as soon as they are committed. Similarly for an overlap of an interactive modify with a rule write, the modifier should be able to access the changes performed by the rule write if he wishes to.

To summarize, the user will be exposed to the following in case of concurrent access.

- Read-Modify Conflict - A reader is always allowed to read. The most current version of the object is returned. A reader is made aware of a concurrent modify as soon as it is known to the system.
- Modify-Modify Conflict - The modifier is made aware of a second modifier once it is known to the system. It is the responsibility of each modifier when he commits to merge his modifications with any version received since his modify began. This guarantees that all relevant changes committed so far are accounted for in the latest version of the object.

- Locking - Locking may be explicitly done to prevent modify-modify conflicts. This is done in cases where the modifier feels his changes are vital. Locking an object will not block readers who are always permitted to read earlier versions.

- Locking - Locking is implicitly done by the system when rules are executed. Category 1 rules are distinguished from category 2 rules in that the conflicts in category 1 rules are treated as appends and hence need not be blocked.

User Interface

Notification of the existence of another modifier may be done by greying the object form. The user reading or editing the form is hence aware of other modifier.

A user may be prompted to refresh his form by blinking the refresh command on the object form.

Adding links to an opened folder (i.e., an append transaction) may be done automatically by the system.

Implementation

A distributed Object Lens environment with multiple representatives of objects is assumed. An object manager at each site controls access to objects at that site. The object manager at each site maintains the following information for each object:

- Locked or Unlocked.
- Being Modified, by whom and at what site.
• Being Read, by whom and at what site.
• Timestamp of the latest version when it was created.

When an object is accessed in modify mode, a request is made to the Object Manager. If the object is locked, access is delayed until lock is released. If not, the Object Manager verifies if the object is being accessed (read or modify) by another user. If the object is being modified, the object is presented with an indication that it is being modified. The user is hence aware of the other modifiers. The object manager then broadcasts to other sites that "user" at "site" is modifying object "x", so that other object managers are kept up to date as to who is modifying the object. They will in turn notify local users of a new modifier. Once the user decides to commit his changes, the object manager verifies that no new versions have been received. If a new version has been received, the user is informed at commit time. The user may then explicitly and interactively merge the two together. The committed version will include changes present in the received version. The object manager installs this as a new version with a global timestamp generated at commit time. The version is broadcast to all sites that contain representatives of object "x". Receiving sites will remove "user" at "site" from the list of modifiers for object "x". The new version will be installed as the latest version of object "x". If the object "x" is being locally modified a note of the received version is made.

Similarly for a read, the object manager checks if the object is being modified and if so a shaded form of the object is presented to the requester. If an object manager is aware that someone else is modifying an object that is being read then the user is made aware of that.

The information maintained by each object manager for concurrency purposes is not large since a user will have at most 10 object forms opened at his workstation at any one time.

Concurrency control for automatic transactions can be integrated into our hybrid implementation by adding the following features:
• Using implicit locking to force the serialization of rule

• Deadlock avoidance by preordering locks.

• Using the latest committed version of the object for reads in rule execution.

Granularity of Locking

Locking is best performed at the object level. Reducing the granularity to the field level results in improved concurrency but incurs a large overhead in lock table maintenance. Locking 10 objects with ten fields requires a 100 locks. Moreover, since a large degree of the concurrency control is performed by version control, the improved concurrency due to decreased granularity is not considerable enough to justify the cost.

Primary Copy Locking

Locking is performed using the primary copy approach. The choice is geared by ease of implementation. The copy of the object at the creator's site would be designated as the primary copy. Locking requests are directed to the primary site. Lock conflicts are detected there and relayed back to the requesting object manager. Updates are directed to the primary site and then propagated from there to all other sites that contain representatives of the object. For locked objects, this is efficient since the update is sent with the lock release request.

In case of version control, updates are propagated distributedly to the relevant sites. It is more appropriate not to have a central controller in the laissez-faire case.

To summarize, our proposed hybrid locking and version control scheme accommodates both interactive and automatic transactions. Objects may be locked by the user if he feels his modifications are important or by an executing rule. Users are always able to read old versions of an object even if another user is concurrently modifying the object or if the
object is locked. Rules are not able to read an object if it is locked by another rule or by the user.
Chapter 7
Conclusion

7.1 Summary of Work

We have in the earlier chapters examined the various approaches to building Distributed Object Lens. The four main issues we were concerned with were:

- Distributed Object Lens Architecture
- Creation of Shared Objects in Distributed Object Lens
- Protection of Shared Objects in Distributed Object Lens
- Concurrency Control and Update Propagation in Distributed Object Lens

For each of these issues we will present the alternatives and summarize the tradeoffs. We will then present our recommendations for the best option for Distributed Object Lens.

Distributed Object Lens Architecture

The three options we examined were:

1. Message-Based System
2. Centralized System
3. Distributed System

The message-based system described an object initiation scheme that fits well with Object Lens's extensive use of messages. However, it fails to be comprehensive enough to support a system-assisted update propagation and concurrency control scheme. The system might result in an unaccountable number of object copies being present.
The centralized option is attractive because of its simpler concurrency control scheme and simpler rule resolution scheme. The first claim is valid only if we decided on a traditional concurrency control scheme (e.g., locking or timestamping). The concurrency scheme most suitable for Distributed Object Lens was the proposed hybrid scheme of locking and versions. Our scheme assumes that more than one copy of the object exits in some state or another at one time. This leaves us with the centralized with caching option. In fact, the "centralized with caching" option is equivalent with respect to our proposed concurrency control scheme to the distributed option with primary copy update. Despite the improved performance that can be attained by caching, the bottleneck imposed on the system by the central site still exists. The centralized cached approach implies a central rule resolution paradigm. All rules are resolved at the central site, and the objects that match are returned. Note, however, that even in the centralized case, private objects are located at the local workstation, hence any rules that consider shared as well as private objects need to be applied at the local workstation for the personal objects and at the central site for shared objects.

Much of the complexity in the distributed approach is due to complex concurrency scheme and complex rule resolution scheme. As mentioned earlier performance of the concurrency scheme is somewhat the same for the centralized with caching and the distributed approach. The rule resolution paradigm on the other hand is complex because objects are distributed over the system. Fully redundant replication will eliminate this problem but introduce a large overhead to propagate updates and a large increase in the amount of memory space needed.

We have eliminated concurrency control as a key consideration in deciding between the "central with caching" and the "distributed approach". The key considerations that remain are rule resolution and contention. Rule resolution may be simplified in the distributed
approach if we consider the fully redundant case. The fully redundant case requires more memory space to store at each workstation all objects used whether personal or shared.

To summarize, the key issues in deciding between the distributed and centralized approach are:

<table>
<thead>
<tr>
<th>Centralized with caching</th>
<th>Distributed Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottleneck</td>
<td>No contention at central site</td>
</tr>
<tr>
<td>Simple Rule Resolution</td>
<td>Complex Rule Resolution</td>
</tr>
<tr>
<td></td>
<td>Simplified in Fully Redundant Case</td>
</tr>
<tr>
<td></td>
<td>Tradeoff between &lt;complex rule resolution&gt; and &lt;Increased memory Space and Increased Locking and Update Propagation Overhead&gt;</td>
</tr>
</tbody>
</table>

**Creation of Shared Objects**

We proposed two ways of making an object shared. The originator can:

1. insert a reference to it in a Mail Message.
2. insert a link to it in a shared object.

We concluded that Distributed Object Lens needed to support both methods. The first was essential to initiate sharing of a parent object. The second was natural for sharing of child objects.

**Protection of Shared Objects**

The two options we examined were:

1. Capability-Based System
2. Access Control List System
The proposed scheme is:

- Hybrid of Capability-based and Access Control List.

A capability-based protection mechanism is the natural choice for Distributed Object Lens. Object Lens is a ticket-oriented system in the sense that users need to have links (unique object ids) to objects before they can access them. It is thus reasonable to augment the unique object id with the relevant fields indicating read, write and delete access. Capability-based system has its pitfalls (uncontrolled access, revocation of access).

An access control list system would mean a tremendous overhead in maintaining lists for each object. A permissive list would be too long to maintain.

We propose to use access control lists in a restrictive manner to support the capability-based system. This hybrid system allows users access to objects if they have the correct access rights as indicated in the unique object id and if they are not restricted by the list. The overhead incurred here would be less than the permissive case mainly because of the size of the list. The restrictive list would be easier to maintain. The list is specified by the creator by indicating authorized users by username or by formulating a set of rules that specify authorized users' attributes.

**Concurrency Control and Update Propagation**

The four options examined were:

1. Simple Two-Phase Locking
2. Basic Timestamping
3. Distributed Certification
4. Version Control
The proposed scheme is:

- Hybrid of locking and version control.

Users in Distributed Object Lens interactively edit objects for long periods of time.

Two-Phase Locking resolves conflicts by allowing at most one user at a time to modify an object. This means that the second user has to wait (perhaps for a long time). Locking is thus an unreasonable approach.

Basic Timestamping scores worse since the user is not aware of other concurrent users. The user is notified that his work might result in a conflict and thus rejected when he is ready to commit the changes. Such unfriendliness would not be tolerated by an Object Lens user.

Distributed Certification scores badly as well. The user faces the same unfriendliness as the Timestamping approach. It is at commit time that a user's modifications are rejected because of a conflict.

Version control's main attractiveness is its laissez-faire attitude. Concurrent users are free to make their modifications and hence create new accessible versions of the object. These versions can be then merged to create the new current object. This scheme however fails to resolve serializable conflicts (modifications that are dependent on a concurrent users' work).

We hence proposed a hybrid scheme that combines locking (explicit), notification and version control. Explicit locking is provided for serializable conflicts. It is the user's responsibility to assess his changes as vital and hence lock the object making it inaccessible.
to others. If the object is already locked, then other users wait, trusting that changes made to the object are vital. Notification warns users of other concurrent users in the laissez-faire environment. This modification might lead to informal communication between the users to determine any possibility of conflicts. At commit time, the user is prompted to merge any pending versions that have been received since the user began his work.

7.2 The Proposed Distributed Object Lens

We choose to describe the distributed approach to implementing Distributed Object Lens because of the generality of the solution. A special case of this is the "centralized with caching" approach. The system is depicted in Figure 7-1.

![Diagram of Distributed Object Lens](image)

**Figure 7-1: Distributed Object Lens**

Shared objects would not be replicated at each site, however several object instance representatives of an object would exist in the network, probably at sites where they are most used. Hence, some object access would be remote and others would be local. We might in some cases choose to make this apparent to the user. System wide object ids are indicative of the object's location, the version time of creation and the access rights to the object. Since we allow replication, a catalog is maintained at each object manager
indicating the other locations of representatives of shared objects stored locally. This is
needed to ensure that updates are propagated to those sites. Updates (new versions) are
propagated by the site where the update occurs. Distributed update propagation is used in
case of version control and primary copy locking and update in case of locking.

The Object Manager will have the following duties:

- Translation table of the system wide ids generated by the object manager and
  the internal ids indicating the physical location of the object in memory
generated by the local object server.

- A list of all objects currently being viewed at the local workstation.

- A list of all locked objects.

- A list of any received versions of objects being viewed.

- A catalog of local shared objects and their other sites.

- The capability to reduce access modes of an object given the system wide
  object id.

- Maintain restrictive access control lists for shared objects at the local site.
  These need to be replicated at each site where a copy of the object exists.

---

3This catalog could be maintained centrally for all object managers

4Again such a list could be maintained centrally to minimize on space used and update propagation in case
   of change
7.3 Direction of Future Work

We feel that in our quest for an object sharing scheme for Object Lens we have shed some light on the relevant features that influence the design of a concurrency control scheme in an object-oriented hypertext system that incorporates both user-driven long transactions and short-lived automatic transactions. Our scheme differs from those proposed earlier in that it allows for replicas and versions of an object to exist. It also permits a user to always view the most recent version of an object. We propose that it is enough to warn the user of a potential conflict and leave it to the user to resolve the conflict by merging relevant versions together. It is worthwhile to explore the possibilities of automatic merging by the system using a specific set of rules.

Initiating sharing using mail messages is another feature not common in hypertext systems. This is characteristic of our system because of its integration with an electronic system.

Protecting objects is an issue that has not been seriously addressed in the hypertext community. We feel that this is important in a distributed environment where object sharing is possible.

Our examination of object sharing in Object Lens reveals a class of information sharing and collaborative applications which have less stringent concurrency and consistency requirements for some transactions and objects they support. We identified those transactions as long interactive transactions that involve editing object forms. Multiple users in such applications need not see consistent copies of the object as long as they are aware of the existence of other versions. Moreover, a user's role has expanded to include explicit merging of different versions. These applications need to also support short automatic transactions that require to see consistent copies of the object. Concurrency
control is much more stringent in this case. It is not sufficient to provide concurrency control for one set of transactions or another, but rather our system must provide concurrency control schemes to accommodate both types of transactions.

The benefits of a less stringent consistency and concurrency requirements indicate that future cooperative work applications should allow workgroups to choose between strict concurrency control provided by locking and more flexible concurrency control provided by version merging.

This work has left out many of the implementation details. It was successful in presenting the design space for Distributed Object Lens and in recommending schemes for object creation, object protection, concurrency control and update propagation. The true test of these schemes is in the actual implementation of the Distributed Object Lens prototype. Depending on the choice of a platform for Object Lens a more detailed analysis of the following needs to be performed in the following area during the implementation phase:

- Object Deletion.
- Distributed vs. Primary copy update in the case of version control. We have outlined a distributed option which could simply be specialized to the primary copy option.
- Rule resolution.
- Recovery.
- Version management.

The power of cooperative work applications is only realized with the availability of object sharing. We hope that the first prototype of Distributed Object Lens would illustrate the importance of semiformal systems to achieve sharing of semistructured knowledge and information.
References

[Berenson 89] Richard Berenson
Object Lens Architecture.
January, 1987
Internal Lens Group Memorandum.

[Bernstein 80] P.A. Bernstein and N. Goodman
Concurrency Control in a System for Distributed Databases (SDD-1).

[Bernstein 81] Philip Bernstein and Nathan Goodman.
Concurrency Control in Distributed Database Systems.
*ACM Computing Surveys*, June, 1981.

[Bobrow 86] Daniel Bobrow and Mark Stefik
Object Oriented Programming: Themes and Variations.

[Carey 88a] Michael Carey and Miron Livny

[Carey 88b] Michael Carey and Miron Livny

[Ceri 84] Stefano Ceri and Giuseppe Pelagatti.
*Computer Science Series: Distributed Database: Principles and Systems*.

Version Management in Gypsy.

[Conklin 87] Jeff Conklin.
Hypertext: An Introduction and Survey.

[Coulouris 88] George Coulouris and Jean Dollimore.
[Decouchant 86] D. Decouchant.
Design of a Distributed Manager for the Smalltalk-80 System.
OOPSLA '86, September, 1986.

Contexts : A Partitioning Concept for Hypertext.
Proceedings of the Conference on Computer-Supported Cooperative

[Demers 88] A. Demers, D. Greene, C. Hause, W. Irish, J. Larson, S. Shenker,
H. Sturgis, D. Swinehart and D. Terry.

[Fishman 87] D.H. Fishman, D. Beech, H.P. Care, C. Chow, T. Connors, J.W. Davis,
N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, A. Neuman,
T.A. Ryan and M.C. Shah.
Ins. An Object-Oriented Database Management System.

Semistructured Messages are Surprisingly Useful for Computer-
Supported Coordination.

[Gref 87] Robert W. Gref.
Distributed Database Systems: A Comparison between ORACLE and
INGRES.

[Greif 88] Irene Greif, Leonard Kawell, Steven Beckhardt, Timothy Halvorsen and
Raymond Ozzie.
Proceedings of the Second Conference of Computer-Supported

[Haas 82] L.M. Haas et al.
R*: A Research Project on Distributed Relational DBMS.
Data Base Engineering, October, 1982.

[Hornick 87] Mark Hornick and Stanley Zdonick.
A Shared, Segmented Memory System for an Object-Oriented Database.

A Survey of Techniques for Synchronization and Recovery in
Decentralized Computer Systems.

Object Lens: A Spreadsheet for Cooperative Work.
Time, Clocks and the Ordering of Events in a Distributed System.

[Lee 88] Jintae Lee and Thomas Malone
How Can Groups Communicate When They Use Different Languages

[Maier 87] David Maier and Jacob Stein.
Development and Implementation of an Object-Oriented DBMS.

Adding Semiformal Structure to Hypertext.

An Overview of the Architecture of the Distributed Database System POREL.

A Distributed Object Server.

[Purdy 87] Alan Purdy and Bruce Schuchardt.
Integrating an Object Server with Other Worlds.

[Reed 78] D. P. Reed
Naming and Synchronization in a Decentralized Computer System.

[Reps 88] Thomas Reps et. al.
Support for Integrating Program Variants in an Environment for Programming in the Large.

[Rowe 86] Lawrence Rowe.
A Shared Object Hierarchy.

[Saltzer 75] Jerome Saltzer and Micheal Schroeder.
The Protection of Information in Computer Systems.
Timestamp Based Certification Schemes for Transactions in Distributed
Database Systems.

Concurrency Control and Consistency of Multiple Copies in Distributed
INGRES

The Design of POSTGRES.
In Proceedings of the 1986 ACM-SIGMOD International Conference
on Management of Data, 1986.

A Majority Consensus Approach to Concurrency Control for Multiple
Copy Databases.

Supporting Collaboration in Notecards.
Proceedings of the Conference on Computer-Supported Cooperative