Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

CONCURRENCY CONTROL AND REMOTE SHARING FOR A REPLICATED COLLABORATIVE ENVIRONMENT

E. H.-Y. Wu

September 1991
CSD-910065

UNIVERSITY OF CALIFORNIA
Los Angeles

Concurrency Control and Remote Sharing for a Replicated Collaborative Environment

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science

by

Elsie Hung-Yun Wu

1991

TABLE OF CONTENTS

1 Introduction **1**

2 Related Work **5**

2.1 coSARA Database Management and Data Sharing 5

2.2 Consistency and Concurrency Control 6

3 Proposed coSARA Concurrency Control **9**

3.1 Dependency Detection with Locking (DDL) 10

3.1.1 The Model 10

3.1.2 DDL Conflict Detection with No Message Loss 13

3.1.3 DDL Conflict Detection with Message Loss 16

3.2 Proof of Concept 17

3.3 DDL with WYSIWIS 21

3.4 DDL Transactions 22

3.5 Complex Objects 26

3.6 Recoverability and Transaction Abort 28

3.7 DDL Optimization 29

3.8 Overhead of DDL 30

4 DDL and Remote Sharing: The Implementation **32**

4.1 Properties of Objects 32

4.2 Properties of Operations 39

4.3 DDL Locking Enforcement 44

4.4 DDL Conflict Resolution 45

4.5 Network Support for Updating Remote Replicas 48

5 Conclusion **53**

5.1 Extensions to DDL 53

5.2 Extensions to Communication Support 54

5.3 Extensions to coSARA Object-World Database Management 54

A Scenarios of Conflict Resolution **56**

A.1 Scenario of Update Conflict Resolution 56

A.2 Scenario of Lock Conflict Resolution 59

References **61**

LIST OF FIGURES

3.1 Operations of DDL with a Single Object 13

3.2 DDL Detects Concurrent Updates 15

3.3 DDL Conflict Detection with Message Loss 16

3.4 Possible Transmission Scenarios 18

3.5 Starting Point for Update 18

3.6 Messages Out of Sequence 19

3.7 Independent Updates with n Updating Sites 20

3.8 Lost Updates with n Updating Sites 21

3.9 Transactions with Dependency Detection Algorithm 22

3.10 Impact of WYSIWIS on Transactions 23

3.11 Tree Locking and Intention Locking 26

4.1 CLOS Definition for Class Counter and Method Increment 39

4.2 Method Increment Which Updates All Replicas 39

4.3 The Broadcast Version of Increment 41

4.4 The :after Method of Increment 42

4.5 The Non-broadcast Version of Increment 42

4.6 DDL Locking Enforcement 44

4.7 Using Copy-In-Place to Resolve Update Conflicts 47

4.8 Updating Remote Replicas with No Broadcast 49

4.9 Updating Remote Replicas with Broadcast 50

4.10 Updating Remote Replicas with Broadcast on LAN Only 51

A.1 The "workday" and "calendar" Objects 57

A.2 State of "Tuesday" at Time of Update Conflict 58

A.3 Window Notifying Users About the Update Conflict 58

A.4 Window Notifying Users About the Lock Conflict 59

ACKNOWLEDGEMENTS

I wish to express my most sincere gratitude to my committee chair, Professor Gerald Estrin, for his guidance, encouragement, and insightful comments during the course of my studies. He has given me the opportunity to grow and helped me in the process. He is the best advisor, in all aspects of life, one can ever hope to find. I wish to thank Professors Mario Gerla and Richard Muntz for serving on the thesis committee. I have gained tremendously from their teaching. I am indebted to Steven Berson, Yadran Eterovic, and Ivan Tou whose wisdom and constructive criticism have been invaluable. I wish to express special thanks to Steven for painstakingly helping me with TCP/IP and numerous other aspects of my thesis.

I would like to thank my grandfather and my parents for their love and support throughout the years. I would never have made it this far without their sacrifice. Much credit goes to Charles and Terry for their emotional support. Finally, special thanks goes to Eric who made this experience most enjoyable.

This research on collaborative design environments has been supported, in part, by the Defense Advanced Research Projects Agency, Hughes, Interactive Development Environments, IBM, NCR, Perceptronics, Sun Microsystems, TRW and the University of California through its MICRO Program.

ABSTRACT OF THE THESIS

Concurrency Control and Remote Sharing
for a Replicated Collaborative Environment

by

Elsie Hung-Yun Wu
Master of Science in Computer Science
University of California, Los Angeles, 1991
Professor Gerald Estrin, Chair

Database consistency is of major importance in cooperative systems. To provide sharing at the *What You See Is What I See* level, and to minimize time spent on remote data access and delay due to network traffic, many systems use data replication. A serious problem in this context is how to maintain mutual consistency while providing the high degree of concurrency required of cooperative systems.

When users are collocated and are working on the same network, mutual consistency can be preserved by taking advantage of the built-in broadcast facilities which guarantee that every site will see the same sequence of updates. The responsiveness and richness of face-to-face communication further enhance the coordination of data access. However, circumstances may require cooperative sessions to span several networks, hundreds of miles apart. Since users could then no longer negotiate for data access synchronously, several users might seek to update the same data item simultaneously, leading to update conflict. When data packets must go through bridges and gateways, inconsistency due to network failure is not unlikely. In this paper, a model is proposed to detect all inconsistencies caused by lost messages, messages received out of sequence, and multiple independent updates on a common data version. Because it treats lock requests and releases as updates on the data item, it is also able to detect lock conflicts. A key feature of this protocol is its ability to provide a transparent interface across all modes of data sharing – on the same network or on interconnected networks; cooperatively without locking or coordinated with locks. The only noticeable difference lies in the means of conflict resolution.

CHAPTER 1

Introduction

"Until recently, computer-based collaboration between geographically dispersed users has been limited primarily to electronic mail" [LANT86]. Inherently, electronic mail imposes significant delays on any group discussion, thus impeding productivity of cooperation. Even though multimedia conferencing and FAX are available to speed up the process of cooperation, they are limited by the lack of editing capabilities. To support more efficient and effective cooperation, researches have begun in areas of Computer-Supported Cooperative Work (CSCW). The goal of CSCW is to provide real-time, interactive, and distributed communication among users, regardless of the physical distance between them [ELLI91]. There is a fundamental need for computer-based conferencing systems which enable conferees to exchange and manipulate graphical and textual data via computers, and to coordinate their interactions through teleconferencing support.

Database consistency is of major concern in such cooperative systems. Users must be able to share data without corrupting the data. To provide sharing at the *What You See Is What I See* (WYSIWIS) [FOST86] level and to minimize time spent on remote data access and delay due to network traffic, many systems use data replication. A serious problem in this context is how to maintain mutual consistency while providing the high degree of concurrency required of cooperative systems. The advantage of replicated cooperative systems over conventional replicated systems lies in the users' ability to coordinate data access, hence reducing the possibility of update conflict.

coSARA (collaborative System ARchitects' Apprentice) is a specialized cooperative environment for computer system designers [MUJI91]. The motivation for its predecessor – UCLA System ARchitects' Apprentice (SARA) [ESTR86] – rose from the increasing complexity of computer systems. Its goal was to investigate the extent to which computer-based tools could extend the capability of computer system designers in realizing intended systems. For this purpose, SARA research created an environment needed by system designers and analysts, adding to it computer support for design methods used throughout the design process. Although this was a very powerful concept, it had its limitations, for it supported only interaction between individual designers and the system. Collaboration among designers was supported through a centralized file system. As computer systems become more complex, the support for effective teamwork and for designers to collaborate interactively on the design is essential. The concept of SARA has therefore been extended to a design

environment for a team of designers ***collaborating interactively through computer systems*** to produce a desired system. This collaborative environment is coSARA. The original approach of SARA, an interactive design aided by computer systems, is retained and extended to facilitate collaboration.

The ideal coSARA can be described as follows: A group of designers collaborate interactively in a conference style setting. The location of the designers is not important – these designers can be sitting round-the-table in a conference room or can be located remotely from each other, forming a "virtual" table with other designers. Rather, the method of collaboration is the key issue. Each designer has at his/her disposal an interactive display. When a designer joins an ongoing meeting, this designer can view the same objects on his/her display as other designers present at the meeting. These data objects can be modified by one designer at a time, while the changes will be reflected, as they are made, on the displays of other designers who have also joined this ongoing meeting. In other words, data are displayed in a "WYSIWIS" fashion, assuming that the designers are looking at the same object.

In the current coSARA environment in which users are collocated and are working on the same local area network, data consistency is preserved across all sites at all times by forcing all participating sites to listen to the same broadcast channel. Upon receipt of a broadcast message, each site processes the data according to the message. All sites receive and execute updates in the order in which update messages are put on the broadcast channel. At most one update message can be on the broadcast channel at any given time. As a consequence of these conditions, consistency is guaranteed. An added advantage to this purely face-to-face collaboration environment is the responsive and richness of face-to-face communication. Such communication is rich in that many unspoken thoughts may be displayed via gestures and facial expressions. These qualities further enhance the coordination of data access and reduce the chance of users corrupting each other's data.

However, the assumption that all collaborators are collocated is too restrictive. Indeed, a collaborative session may span several networks, hundreds of miles apart. In such an environment, coordination of data access can be done with the aid of multimedia conferencing, though, because of the geographic distance, the response is subject to delay. Since users could then no longer negotiate for data access synchronously, several users might seek to update the same data item simultaneously, leading to update conflicts. Further, since data packets would have to go through network bridges and gateways, we could no longer take for granted that all updates will be received and hence executed in the same order at all sites. Means must be provided to maintain database consistency while supporting the high degree of sharing necessary in a cooperative environment. When update conflicts do arise, means must be provided to aid conflict resolution.

Database management in the coSARA environment is different from that in conventional database systems. First, data in coSARA exist as objects; they have prescribed encapsulation

and inheritance properties [DITT86, DIED89, NIER89]. As such, data can be structured as complex objects, hierarchically containing other (possibly complex) objects. Each complex object may be accessed as one entity, or as separate entities each being an object component. Therefore, coSARA database management must support sharing and locking of complex objects.

Second, to reduce the amount of remote data access, data objects in coSARA are replicated on demand. There is a tradeoff between full-replication and replication on demand. If the space of shared data is large, then full-replication reduces network traffic. However, the amount of memory needed at each site is significant. The argument used at coSARA, and a valid one, is that special users of the coSARA environment – the tool designers – should be given the flexibility to choose the scheme they want to use. If they believe that replication on demand is more appropriate for the tool, then they can replicate data objects on demand. On the other hand, if they prefer full-replication, then the tool can be built in such a way that when it is started, it automatically gets a copy of all existing objects, thus working with a fully replicated database. This flexibility is unique to the coSARA implementation.

Third, to maintain database consistency, each update is broadcast to all sharing sites. In coSARA, this is done by broadcasting the name of the update function and its arguments instead of broadcasting the output of the update. There is a tradeoff between the two schemes. By broadcasting the output, we could ensure that the database is consistent even in failure (by majority vote). In addition, if the update function requires extensive computation and generates relatively little output, then it would be more efficient to broadcast the output, instead of having each site independently compute the result. However, since data in coSARA are highly graphical, each update is likely to generate voluminous output, hence network traffic. By broadcasting the function name and its arguments, less data needs to be transmitted per update. This allows us to take advantage of the datagram service and its ability to adapt to network failures and congestions [TANE89].

Fourth, coSARA is an environment for system designers. Transactions operating on these objects might last an arbitrarily long time. Although we would reasonably limit any coSARA collaborative session time to be less than an eight-hour work day, it is undesirable for one user transaction to have exclusive access to a large, complex object (and hence its components) for such an extended period of time. coSARA needs to support long-duration transactions while maintaining a high degree of sharing.

Finally, the goal of coSARA is to provide object sharing on a WYSIWIS level, regardless of the physical distance between coSARA users and whether locking is used to restrict access. Hence, all updates – including those locked for exclusive access – must be immediately broadcast to other participating sites, so as to maintain the same view at all sites. Conventional locking protocols have no provision to allow reading of dirty data for WYSIWIS. Means must be provided to support locking while preserving the WYSIWIS notion.

In this paper, I propose a model which guarantees to detect all database inconsistencies caused by lost messages, messages received out of sequence, and multiple independent updates on a common data version. It treats lock requests and releases as updates to the data item; therefore, it detects lock conflicts in the same manner. A key feature of this protocol is its ability to provide a transparent interface across all modes of data sharing – on the same network or on interconnected networks; cooperatively without locking or coordinated with locks. The only noticeable difference lies in the means of conflict resolution.

This paper is organized as follows. Chapter 2 describes the current configuration and status of coSARA and briefly surveys previous research on consistency and concurrency control for replicated or cooperative systems. Chapter 3 describes the proposed consistency detection model and proves that the model detects all update and lock conflicts caused by lost messages, messages received out of sequence, and multiple independent updates on a common data version. Chapter 3 also discusses how this model deals with the characteristics (e.g., WYSIWIS, richly hierarchical data structures, long-lasting transactions) that make coSARA database management unique. Chapter 4 describes the implementation of this model in coSARA, and how updating of remote sites is supported. Chapter 5 concludes this paper with presentation of open issues and recommended future directions.

CHAPTER 2

Related Work

2.1 coSARA Database Management and Data Sharing

As mentioned earlier, data in coSARA exist as "objects" and are replicated on demand. The coSARA "object-world" is the set of operational primitives that support the management and the sharing of these data objects. The current implementation of the coSARA object-world has three functionalities. First, it is the database of objects. This database is common to all the designers, such that design objects created by designers and design objects provided by the system can be shared and easily modified by all collaborators through a uniform interface. To ensure that all objects can be shared across sites and stored in the persistent database, each coSARA object has a unique and immutable identifier (the "storable-id") used for internal references. To the user, each object is identified by its type and name (the "storable-name"). To ensure that each object can be correctly accessed at any time and from any site, there is a one-to-one mapping between each object and its identifier.

Second, the object-world provides a set of primitives to create, access, save, and delete objects in the database. This set of operational primitives differs from the conventional database primitives in the sense that replication on-demand must be supported. When an object is created, all collaborating sites must know about its existence and its whereabouts so that, when desired, it can be easily accessed and replicated. When an object is saved, measures must be taken to ensure that it can be later retrieved from any other site, regardless of its physical location. When an object is deleted, measures must be taken to delete all replicas of the object, regardless of their whereabouts, and to remove all references to the replicas.

Third, the object-world provides a set of facilities to support communication between collaborating sites and propagation of updates on a local area network. It manages the creation and deletion of communication sockets [COME88]. It handles the sending and receiving of messages between any two sites, as well as the broadcasting of messages on the local area network. Any two sites can communicate through TCP/IP datagrams or stream connections [COME88], requesting information on objects, announcing the creation of new objects or deletion of obsolete objects, etc.

Note that three important functionalities were missing in the implementation of the

coSARA object-world prior to this thesis work. First, no facilities were provided to ensure the consistency of the database. Second, no facilities were provided for concurrency control. Third, no facilities were provided to broadcast on interconnected networks. Unless the first two are provided, database consistency is vulnerable to site or network failure and to uncoordinated access to the same data object by more than one user. Hence, users are likely to work unknowingly with an inconsistent database. Unless the third functionality is supported, users cannot share data across interconnected networks and hope to maintain the same view of the database at all times.

The coSARA object-world is built on Common Lisp Object Systems [KEEN89, STEE90] – an object system integrated in Common Lisp [STEE90].

2.2 Consistency and Concurrency Control

Concurrency control is the means provided by any DBMS to prevent uncoordinated access to the same data item by more than one user [KORT86, BERN87, ULLM88]. In a face-to-face cooperative environment, such coordination of data access is simplified by the human interface – through delegation of tasks and responsibilities and through active negotiation. When users are not collocated, on the other hand, the ability to actively negotiate for object access is reduced. Hence, concurrency control must be provided to allow users to perform tightly coupled activities without tripping over each other.

Conventional locking protocols have often been criticized as overly restrictive for cooperative environments. A lock is an access privilege to a single item that must be placed on the item before reading or writing it. If a user program tries to lock an already locked item, the program is blocked and may not continue until the lock is released. While any number of transactions can hold a read-lock on the same item at the same time, write-locks are exclusive. In other words, when some transaction holds a write-lock on an item, no other transaction can obtain either a read- or a write-lock on the item. This exclusion is undesirable for coSARA since participants must be able to share the same view on the database, including those data items locked for exclusive read/write privileges, in a WYSIWIS fashion. Also undesirable is the overhead in obtaining the lock, especially if the data is already locked. Yet a third problem is with the granularity of locking: The larger the granule, the less sharable. This is particularly noticeable when hierarchically structured complex data items, as those in the coSARA environment, are to be shared. Finally, when locks are grouped into transactions, there is the complication of long-lasting transactions. If these transactions are allowed to lock large, complex objects for an extended period of time, sharability is significantly reduced.

Also undesirable for groupware concurrency control are the floor passing strategies. Their flaw is that at most one person can be active at any given time. All other users wait to update

any object until the control of the floor is changed. This is unnecessary and inappropriate for coSARA, where users are allowed to simultaneously work on different pieces of the project during a collaborative session. By limiting the number of active users to one at any given time, productivity is significantly hindered. Performance is worse when remote users are involved in the cooperative process, for floor access negotiation (or even floor passing) will suffer from network traffic.

Dependency-detection by Stefik, et al. [STEF87] is more appropriate for cooperative environments because it allows simultaneous threads of progress. Data are labeled by a stamp describing the author and the time of change. Every request to change data broadcasts the new data, its stamp, and the stamp of the previous version of the data on the originating machine. Upon receiving the update request, each site first checks whether the stamp on the previous version of the data item at the originating site matches the current stamp on the local replica of the data item. If so, the update is processed and the local replica of the data item gets a new stamp from the update request. If the two stamps are different, a "dependency conflict" is signaled, with conflicts resolved by the users. The advantage of this scheme is that non-conflicting operations are performed immediately upon receipt, yielding good response. There are two disadvantages to this scheme. First, it reports *all* conflicts. Consider, for example, a scenario with sites A and B, both holding a replica of object O1. When a site updates O1, it sends the *result* of the update to the other site. Thus, the two replicas are always kept in agreement (assuming no network failure). Suppose now that Site A updates object O1 thrice, but Site B only receives the result of the first and the third update, in that sequence. When B receives the result of A's third update, it reports an update conflict because the stamps on the previous states of the replicas fail to match. It fails to recognize that the result of A's third update actually reflects the result of the second update. If B had recognized that fact and simply ignored the mismatched stamps, then the two replicas would again be in agreement (of course we have to assume that the result of A's second update is not needed anywhere else). In other words, this scheme fails to recognize optimization possibilities for it reports conflicts even for scenarios that could otherwise be automatically resolved. The disadvantage to this scheme is that when remote users are present, conflict resolution may not be easily achieved. This is true with all conflict detection schemes where the semantics of the operation are unbound.

The version vector approach proposed by Parker, et al. [PARK83] also compares stamps on replicas of the data item to detect conflict. Unlike the dependency detection model which uses a simple (author, time) pair as stamp on the data item, the version vector approach uses a set of n (site, version) pairs – called the "version vector" – on each data item, where n is the number of sites holding a replica of the data item. Each time an update on the data item originates at site S_i, the version number associated with site S_i in the data item's version vector is incremented by one. By definition, two version vectors (of the same data item) are compatible when all n (site, version) pairs of one vector is at least as large as all n (site, version) pairs of the other vector. Then, the data item is mutually consistent across all sites when one vector is at least as large in every vector component as any other

vector in any site with a replica of the object. The replicas conflict otherwise. Whereas the dependency detection model reports conflict when two previous stamps fail to match, the version vector approach goes a step further to determine if one replica is an earlier version of another replica. If so, the site with the earlier version can simply obtain a later version, with no loss in semantics. The advantage of the version vector approach over the dependency detection model is precisely its ability to discriminate earlier versions from incompatible versions. However, this scheme is not without flaws. First, when the size of the data item is small but the number of sites holding a replica of the data item is large, a significant portion of the storage would be used for version vectors. Second, as updates are broadcast to other sites, the amount of encoding and decoding to be done on version vectors may hinder performance. Third, since an update broadcast in coSARA is done by broadcasting the function name and its arguments, where each argument contains a version vector, the amount of data to be transmitted for each update can easily exceed the allowable datagram size of 1500 bytes [COME88], causing packet fragmentation, and making update broadcast less efficient.

There are also proposals which improve response time by having each site process the update as it is received. If it is later detected that the updates were conflicting, or executed out of order, the effects will be reversed, and updates re-executed in the correct order. dOPT (distributed operation transformation algorithm) is a scheme which detects such update conflict [ELLI90]. It proceeds without locking or rollback. When conflicts are detected, transformations are performed to reverse incorrect side-effects and to guarantee consistency. The approach relies upon application-specific semantic knowledge of the desired outcome of concurrent operations. Although this scheme would significantly improve the response time, it degrades the notification time, since all updates must be checked for conflicts before it can be processed at the receiving sites. Further, it is inappropriate for coSARA whose output is WYSIWIS and mostly graphical. It is highly undesirable, if not impossible, to have to "reverse" any graphical output from previous operations. This algorithm works for GROVE [ELLI90] because it is a text editor, with simple editing operations that can be performed in a relatively short period.

In summary: Conventional locking protocols are overly conservative and unable to support WYSIWIS. Overhead incurred in obtaining the locks may significantly degrade system performance. Floor passing strategies are inappropriate for coSARA because they limit the number of active users to one at any given time. A dependency detection protocol is more appropriate for cooperative environments because it allows simultaneous threads of progress. However, user intervention may not be the most efficient means of conflict resolution, particularly when remote users are present. The version vector approach is attractive not only because it allows simultaneous threads of progress, but also because it discriminates between earlier versions and incompatible versions. However, the amount of data to be stored, encoded, transmitted, and decoded may insignificantly degrade performance. Finally, algorithms which require that side-effects be reversed in case of conflict are inappropriate for coSARA whose output is highly graphical.

CHAPTER 3

Proposed coSARA Concurrency Control

In the coSARA object-world, data objects are replicated on demand. By storing copies of the object at client sites where they are needed, we reduce the network and performance overhead that may otherwise be incurred with frequent remote access. By replicating the object on demand (only at sites needing the object), storage is more efficiently utilized. The tradeoff is that to maintain mutual consistency across sites, all replicas of the object must agree on one *current* value *eventually.* This means changes to any one replica must be propagated and processed, eventually, at all other sites, in an order which ensures all replicas to achieve the same final value. We define coSARA object-world consistency to be the state in which all replicas of the object agree.

The order in which changes are processed is of major importance in preserving database consistency. Conservatively, all changes should be received in the same order at all sites. A relaxed protocol would require that only interrelated changes be processed in the same order at all sites. When users are working on the same Ethernet network, this order is easily maintained since only one message can be on the broadcast channel at any time, and all messages will be received in the same order at all sites. When users are working on interconnected networks, however, messages may be received out of order and hence special means must be taken to determine the proper order in which changes are to be processed. Needless to say, because of management complexity, it is undesirable to use one data sharing protocol when users are working on the same network, and a second one when users are working from different networks. Further, because a Collaborative Design Environment (CDE) user may join or leave an on-going session at any time, it is also undesirable to force other users to re-initialize their work space every time a participant joins or leaves the session, in order to determine the proper protocol to use. Such a re-initialization is bound to cause a drop in productivity.

Another means to preserve database consistency is to restrict data access to qualified users by concurrency control (e.g., locking). But concurrency control protocols alone will not achieve database consistency in a distributed environment, for the order of message processing still determines the final data values at each site. In coSARA, locking facilities should be provided to CDE users to coordinate access to data items. Users have the freedom of choosing to work cooperatively without locks or in coordination with locks. In either case, update or locking conflicts should be interactively negotiated, for users are the only ones

who have the semantic knowledge to determine the correctness of the database. Although special processing is needed to check for lock conflicts when locks are used, it is undesirable to have to change database interface as a session needs change. Hence, the concurrency control protocol to be used must support both modes of sharing, and provide a transparent interface regardless of locking.

In short, all modes of data sharing – on the same network or interconnected networks; cooperatively without locking or coordinated with locks – must be available to users. It is the users' responsibility to choose the one most appropriate to the session. *The mode of sharing and the interface thereof should be transparent to the users.* The only noticeable difference should lie in the means of conflict resolution. When users are collocated, face-to-face negotiation should be sufficient. When users are geographically dispersed, other types of communication (e.g. multimedia conferencing facilities, FAX, e-mail) may be needed.

3.1 Dependency Detection with Locking (DDL)

The model we propose here is a modification to the dependency detection model proposed by Stefik, et al. [STEF87], extended with locks and treating lock requests and releases as updates. We have chosen to expand on this model for its simplicity. Little data needs to be stored, encoded, transmitted, and decoded per update broadcast. Although this model is unable to recognize optimization possibilities, as pointed out in Section 2.2, this is a cost we can tolerate since recent technology has made networks quite reliable [TANE89].

Because the new model (henceforth referred to as DDL for *Dependency Detection with Locking*) treats lock requests and releases as object updates, it is able to detect lock conflicts in the same manner as update conflicts are detected in the original dependency detection model. Further, this protocol can be used to support all modes of sharing with a uniform, transparent interface. In what follows, we will present the model of DDL, how it deals with locks and transactions, how it supports WYSIWIS, how it facilitates locking of hierarchically structured objects, and its overhead. Most importantly, we will prove that it will detect all update and lock conflicts which may occur in the coSARA object-world.

3.1.1 The Model

For our system configuration, we allow any number of active client sites, each running the CDE client software, and each holding replicas of all objects to be shared. The clients can be on the same network or different networks connected by bridges or gateways. Each site knows the existence of all other sites, and can communicate with all others via message broadcast.

A class *DDL* is defined with two attributes: The first contains the stamp of the last update on an object of class DDL; the second contains all the locks and warnings (the notion of warnings is discussed in Section 3.5) held on the DDL object. All objects in coSARA are of class/subclass *DDL*. This requirement ensures that all objects will inherit these attributes and are able to use any function defined in support of the DDL concurrency control and data sharing.

To preserve coSARA object-world consistency in the current implementation of CDE (where all users are assumed to be working on the same network), when an update is performed on an object, all replicas of the object are updated accordingly at all sites. This update is performed by means of network *broadcast* [COME88, TANE89]. The *method of update* is broadcast to all sites holding a replica, along with the ID of the object to be updated. This object ID is generated when the object is created, and is unique for all sites at all times. All replicas of the object will have the same object ID. The advantage to broadcasting the method of update instead of the result of the update is that there is potentially less to be encoded/decoded by broadcasting the update method and each site does the computation independently.

In the DDL Model, each object is stamped with a unique *update stamp* which identifies the last update performed on the object. This stamp is generated by each client site independently with the aid of a local state counter. With every update, a new stamp is generated, with the value of the state counter attached to the name of the client site. The counter is then incremented, to be used for the next message. This guarantees that all update stamps are unique at all times, across all sites. Note that this method of stamp generation requires no clock synchronization by the client sites. No site needs to depend on another site for its stamp creation.

Each update message contains four essential pieces of information: the ID of the object to be updated, the update function to be performed on the object, the stamp of the previous update performed on the object, and the stamp generated for the new update to be performed. Each update is immediately processed at the originating site by first changing the value of the update stamp on the object. This is assumed to be an instantaneous process since it is merely updating a local replica of the object. It then updates the object and broadcasts the update request to other sites. When another site receives the update message, it compares the stamp of the previous state as shown in the update request with the stamp on its local copy of the object. If the two stamps agree, then the update *depends* on the same previous state as that currently available locally. Only when the two stamps agree can it proceed to update the object according to the update request. If the two stamps disagree, then the originating site must have processed the update on a different state of the object than what is available locally at the receiving site. Such update conflicts are immediately reported at all client sites, so users can resolve the conflict interactively.

Lock requests and releases are treated as object updates in DDL. This is appropriate

because all lock information is actually stored in the objects. Whenever a lock is granted or released, the information in the object will be changed, hence the object is updated. Because lock requests and releases are processed as object updates, they too must be broadcast to other sites for database consistency. In addition, the object will obtain a new stamp to reflect the change. Indeed, when a client site wishes to obtain a lock on an object, it first checks the local copy of the object for lock/warning conflict. If no conflict is detected, it updates the object with a new update stamp; then it optimistically grants itself the lock by changing the lock information in the object. The lock request is then broadcast to other sites with the ID of the object to be locked, the type of lock requested and the lock requester, the stamp of the previous update performed on the object, and the new update stamp. When another site receives the update message (which is actually a lock request), it compares the previous update stamp as shown in the update request with the stamp on the local copy of the object. If the two match, then the update *depends* on the same previous state as what is currently available locally. Further, we can deduce that since no lock/warning conflicts were detected at the originating site, none would be detected at this local site. The update is then processed by recording the lock on the object, with the lock held by the site which sent the lock request. Had the stamps failed to match, a conflict would be reported and update aborted. Lock releases are processed in a similar fashion, although no lock/warning conflicts need to be checked before removing the lock information from each replica.

The requirement that all coSARA objects must be of (sub)class *DDL* is essential in making the mode of data sharing transparent for three reasons: First, with the update stamp attached to each object, we can detect update conflicts (including those from lock requests and releases). This is a way to ensure that all updates for a particular object will be processed in the same order at all client sites, regardless of the network configurations. When they are not processed in the same order, conflicts are reported and resolved by human intervention. Second, by making all objects of (sub)class *DDL*, no special processing is needed in making an object lockable, hence, locking can be readily enforced. Finally, with the lock information stored in each object, the lock requests can be processed as updates, and hence, conflicts caused by lock requests will be detected as usual.

It is important to note here that conflicts will be detected only on a per object basis. As long as updates on each object are done in proper sequence, no conflict will be detected. A sequence of updates on different objects can be performed in different orders at each site as long as the updates on *each* object is performed in the same order at each site. This is so because only the stamps for the particular object being updated are compared for update conflicts. The advantage is that independent threads of progress can proceed at the same time. The disadvantage, however, is that if the update depends on multiple objects (as do transactions), this method will prove to be insufficient. This notion will be explored in Section 3.4.

DDL is considered optimistic because the originating site always assumes itself to be the only site making the update or requesting the lock on that particular object. It also

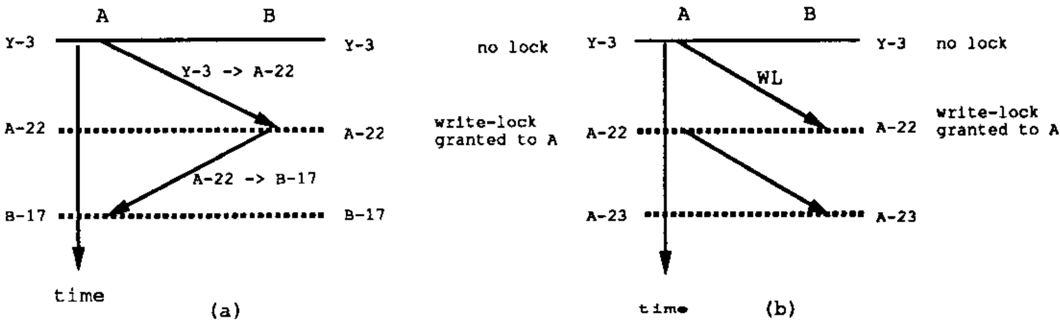


Figure 3.1: Operations of DDL with a Single Object

assumes the local information to be accurate (e.g., in checking for lock/warning conflicts when requesting a lock). This is done to improve both the response time and the notification time. The tradeoff is that when conflicts are later detected, users must interactively negotiate to resolve them.

DDL guarantees that, from a consistent database state, if any update message is lost, whether it be a normal update, or a lock request/release disguised as an update, the conflict will be detected. This is because the lost messages will cause at least a subset of the clients to remain in a *backward* state. When new update messages are received, the clients in the backward state will not be able to match the stamps, and thus report conflicts. Further, DDL will detect any conflict resulting from concurrent yet independent updates at different sites. This is detected because each originating site will be at a more *forward* state than what is expected of it by other originating client sites. Finally, if there are no conflicts, then DDL guarantees that the database will eventually assume a consistent state again.

In the examples that follow, we assume all active clients to have replicas of all objects being updated. This is not always the case since object-world replication is done on-demand. However, the result would still be the same, for each site knows which objects exist on which sites, and update messages are broadcast only to sites holding replicas of the object being updated. Hence, all replicas of the object will be updated accordingly, and sites without a replica of the object will not be informed of the update. When conflicts are detected, they will be detected only by a subset of sites holding the replica of the object, but will be reported at all sites with the object.

3.1.2 DDL Conflict Detection with No Message Loss

Figure 3.1 shows how updates and lock requests are processed in DDL. Two client sites A and B are running, each with a replica of object O1 which is currently being updated. These two sites can be located on the same network or networks connected by bridges or

gateways. In Figure 3.1a, both sites A and B agree on the initial state of Y-3, where Y is the name of the client site which initiated the last update on O1, and 3 is the value of Y's state counter when the update stamp was generated. When Site A wishes to update O1, it first checks for locks placed on O1. If there were any read-locks or a write-lock held by a site other than A, O1 cannot be updated. Let's assume for now that there are no locks on O1. A then changes the update stamp on O1 from Y-3 to A-22, where 22 is the current value of A's state counter. To ensure all update stamps to be unique, A's counter is then incremented to 23 for the next update. A proceeds to update O1 as planned and broadcasts the update request to other sites, in this case, only B. The update request contains not only O1's object ID, but also the update method, the previous stamp of the object (Y-3), and the new update stamp (A-22). When B receives the update request, it compares the state of the local copy of object O1 with the previous state of the object according to the update request. Since B's copy of O1 has not been updated since Y-3, the two stamps match and B proceeds to update the object according to the request. To reflect the update, B changes the stamp on its copy of O1 from Y-3 to A-22. The two sites now have a consistent state with respect to O1. An update by B will be processed in the similar fashion, with a new update stamp B-17. When A receives B's update request (with O1's object ID, the update method, previous stamp of A-22, and new stamp of B-17), A finds the local copy of O1 to be in the same state A-22 as required by B's update request. It therefore updates the object by changing the stamp on the local copy of O1 from A-22 to B-17, and updates O1 according to the update request. Once again, the state stamps on the two copies of the object match, and indeed, the two replicas agree.

Figure 3.1b shows how lock requests are processed in DDL. Recall that lock requests and releases are treated as object updates since the lock information is stored as part of the object. From a consistent initial state of Y-3, user U1 on Site A requests a write-lock on object O1. Site A first checks with the local copy of O1 that the write-lock would not conflict with any lock already placed on O1. In this case, no lock conflict is detected. A then optimistically grants itself the write-lock, and updates the state of O1 from Y-3 to A-22. The lock request is broadcast to B with the object ID of O1, the write-lock mode, lock requester U1, the previous stamp on the object Y-3, and the new update stamp A-22. Upon receiving the lock request, B finds the local copy of O1 to be in the same state Y-3 as required by the lock request. With the previous states matching, we deduce that since no lock conflicts were detected by the originating sites, none will be detected at B. B therefore records in O1 that a write-lock has been granted to A and changes the update stamp from Y-3 to A-22. Until A releases the write-lock, B will not be able to update O1, although all updates on O1 from A will be processed in the same fashion as described above.

A good update conflict detection algorithm should be able to detect conflicts caused by concurrent updates generated on the same object version. Figure 3.2 shows how this is achieved in DDL. In Figure 3.2a, sites A and B independently decide to update object O1 from an initial consistent state of S1 (stamps generated in the same manner as before). Site A updates the state stamp on its local copy of O1 from S1 to S2, while Site B updates the

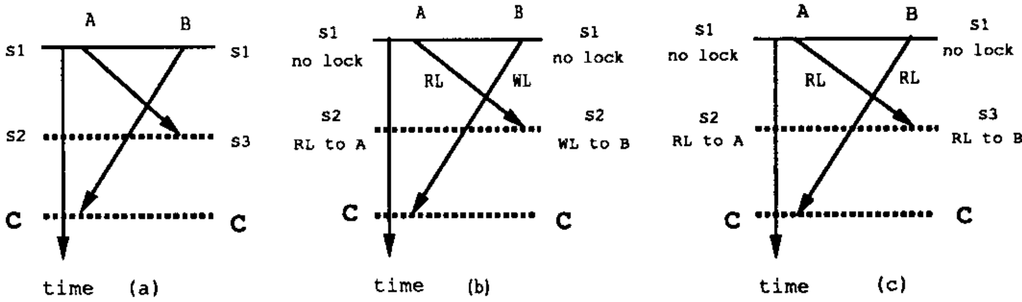


Figure 3.2: DDL Detects Concurrent Updates

state stamp on its local copy of O1 from S1 to S3, both occurring instantaneously. Both sites then broadcast their update messages to the other. When A receives B's update message, it reports an update conflict, because B's update request assumes a previous state (S1) different from what is available at A (S2). Similarly, B reports an update conflict when it receives A's update request which also assumes a different previous state (S1) than what is currently in B's database (S3). Because lock requests and releases are treated as object updates, conflicts will also be detected if the updates are actually lock requests, as shown in Figure 3.2b. A requests a read-lock on O1 while B requests a write-lock. Both sites optimistically change the state stamp on the local replica of the object and each grants itself the lock. When B receives the lock-request from A for a read-lock, B reports a conflict since A's update request assumes a different previous state (S1) of O1 than what is available locally (S3). Similarly, A reports a conflict since it too is at a different state (S2) than what is required by B's update. Indeed, conflict is reported not for the conflict in lock types, but for the conflict in update dependencies.

Certainly, reporting conflict on the mere basis of state stamp mismatch is too conservative, for if two sites concurrently request read-locks on the same object, there is no need to report update conflict, for read-locks are compatible. This notion is illustrated in Figure 3.2c. Indeed, the model can be optimized if sites A and B recognize this compatibility and resolve the conflict automatically without human intervention. Such optimizations will be discussed in Section 3.7.

Our examples here assume only two active client sites. The result can be easily extended to configurations with more than two sites. Suppose N sites are running, two of which independently and concurrently generate updates X and Y based on the same object state. When each site broadcasts its update request (or lock request) to the other N-1 sites, a subset of the N sites is bound to process message X first, while the rest process message Y first. When these sites process the other messages, they will detect the conflict. In fact, the more active sites there are, the more rapidly the conflicts will be detected.

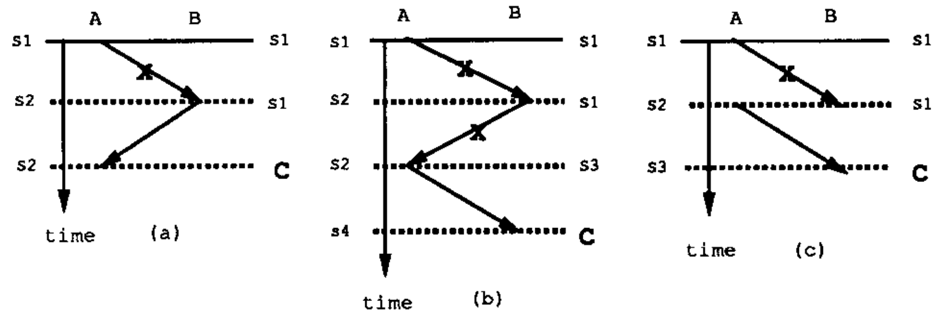


Figure 3.3: DDL Conflict Detection with Message Loss

3.1.3 DDL Conflict Detection with Message Loss

Update conflicts often result from lost update messages, because message loss causes some sites to remain in a backward state while others are properly updated. Figure 3.3 shows how DDL detects conflicts in the presence of message loss. In Figure 3.3a, an update message from A is lost during transmission. Although A has properly updated its local copy of the object and the object state (from S1 to S2), B's copy of the object remains unchanged in S1. When B later updates this object, it assumes the local information to be correct and performs the update on the backward state S1 of the object. B's update request will be broadcast to other sites with S1 as the previous state of the update. A will detect the update conflict since A's local copy of the object is not at state S1, but rather at state S2. This conflict will be reported by A to all the other sites, awaiting human intervention. Conflict would also be reported if the lost message was a request for a read-lock by A. In that case, A would optimistically assume that it has a read-lock on the object, although B has no knowledge of the read-lock. When B broadcasts an update request, A will notice that the previous states of the object at the two sites do not match and report the conflict. This is true even if B's update request is actually a lock request for a write-lock on the object. Hence, regardless of the type of message that is lost, DDL will report the conflict when the previous state of the update fails to match the state of the object locally available.

Conflict will also be detected when more than one message is lost, as illustrated in Figure 3.3b. In this case both sites have updated the state of the local copy of the object, although neither knows about updates made by the other site. When finally an update message is successfully transmitted, the update conflict will be detected because the previous states at the two sites do not match. Indeed, regardless of the number and type of messages lost, DDL will *eventually* detect the conflict.

As pointed out in the last section, reporting conflict merely on the basis of state stamp mismatch is too conservative. Indeed, as shown in Figure 3.3c, this is not always necessary. Here, Site A makes two consecutive updates, the first of which is lost during transmission. B detects and reports the conflict when it receives the second message. If the intermediate

values of the object is not used by B, then it may be more efficient if B can simply copy the final value of the object from A. This inefficiency is more noticeable if the lost update is initiated by A which has a write-lock on the object. The next update by A would be reported as conflict by B because the previous states fail to match. It would certainly be more efficient if B could recognize that A has the write-lock and obtain the most recent copy of the object from A instead of reporting update conflict and awaiting user intervention. Such optimization will be discussed in Section 3.7.

As shown thus far, DDL is able to detect update conflicts for a wide range of sharing: with users working on the same network or on interconnected networks; in a cooperative model (no locks) or with updates coordinated by locks. Each update is done by optimistically assuming the local information to be accurate. However, the protocol is very conservative in that it reports all conflicts to the users instead of trying to resolve the conflicts itself with whatever knowledge it has on the current state of the database.

3.2 Proof of Concept

The goal of this proof is to show that all and nothing but update conflicts will be detected by DDL. Recall that DDL is a conservative protocol which requires all updates to be processed in the same sequence at all sites. This forces all replicas of each object to eventually agree, hence preserving database consistency. It reports an update conflict whenever $S_s \neq S_d$, where S_s is the previous object state at the source site and S_d is the previous object state at a particular destination site. We will therefore prove that *conflict occurs iff* $S_s \neq S_d$.

We divide the proof into two parts:

$$P1: \quad if\ conflict\ occurs\ then\ S_s \neq S_d.$$

$$P2: \quad if\ S_s \neq S_d\ then\ there\ is\ a\ conflict.$$

We will prove each part by contradiction by showing that it is impossible to have a *true condition* and a *false consequence* in each part. We assume m participating sites, n of which update the object ($n \leq m$). Updates from the n sites can be interleaved in any order (an update may follow either from an update by any of the other $n-1$ updating sites or from a previous update by the same site). Further, we make the simplying assumption that a message is either correctly received or not received at all. Hence, four message transmission scenarios are possible, as shown in Figure 3.4. In the first scenario, all updates are received and processed in the same sequence at all m sites; hence, all replicas of the object agree when transmission and updating is completed. This scenario is conflict-free by definition. In the second scenario, sequential update messages have been re-ordered by networks and

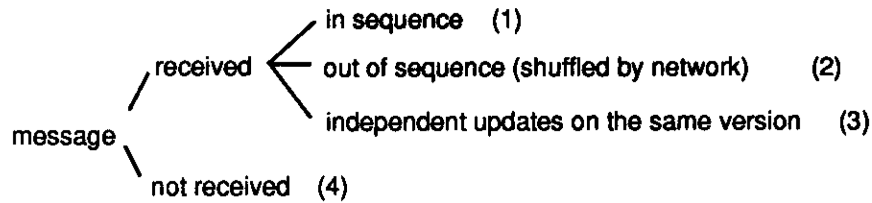


Figure 3.4: Possible Transmission Scenarios

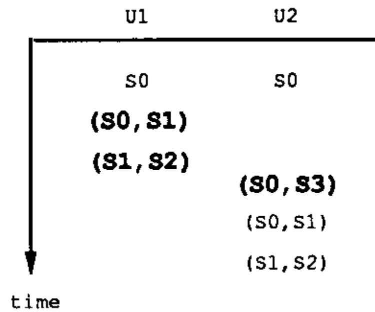


Figure 3.5: Starting Point for Update

gateways. The m sites therefore process updates in varying sequences depending on the order of arrival. In the third scenario, multiple sites independently update the *same version* of an object, thus creating multiple versions of the same object. Such divergence will prevent the database from becoming consistent again. Finally, in the fourth scenario, one or more update messages are lost during transmission; therefore, not all sites receive all updates. The last three scenarios are conservatively assumed to cause update conflicts which must be resolved through human intervention. The first three scenarios constitute all possible scenarios when all messages are received – sequential version update messages are received in the correct sequence, sequential version update messages are received out of sequence, and concurrent version update messages are generated. Since a message can either be received correctly or not at all, the four scenarios make up the set of all possible message transmission scenarios.

To prove *P1*, we assume that there is a conflict and that $S_s = S_d$. As shown in Figure 3.4, only Scenarios 2, 3, and 4 cause update conflicts. Let U be the set of n updating sites $\{U_1, U_2, ...U_n\}$. Let R be the set of non-updating sites $\{R_1, R_2, ...R_{m-n}\}$. We assume that all m sites initially agree on a consistent object state of S_0. Consistent with Scenario 1, we assume that no two sites will independently update the same version of the object (this is considered in Scenario 3). Hence, each updating site $U_i \in U$ must receive all previous updates on the object from all other sites in U before it proceeds to update the object; otherwise multiple versions of the object would be created, as shown in Figure 3.5. Here, Sites U_1 and U_2 initially agree on state S_0. U_1 updates the object from S_0 to S_1 (as represented by the pair (S_0, S_1)) and then from S_1 to S_2. (The boldface type on (S_0, S_1) in U_1's column

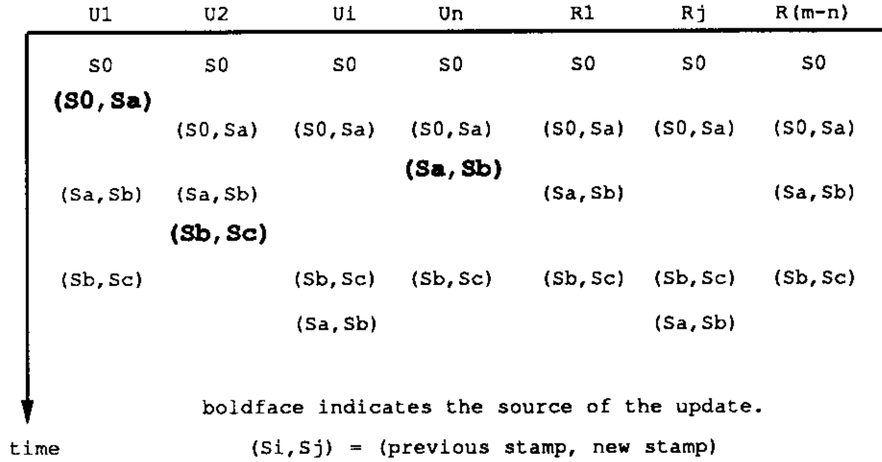


Figure 3.6: Messages Out of Sequence

indicates that update (S_0, S_1) is generated by U_1). If U_2 proceeds to update the object before all previous updates on the object from all other sites are received, then it will update the object from S_0 to some other state S_3. Hence, two versions of the object are created. This is considered as concurrent update on the same object version, and will not be considered now. It wil be considered when proving Scenario 3.

Further, since the correct update sequence is determined by the updating sites, an updating site detects conflicts only when it is receiving updates from other sites, not when it is updating the object. Suppose that the object is updated from state S_a to S_b, and then to S_c, where S_a and S_b are generated from any site(s) in U and S_c is generated by any site in $U - U_i$. If U_i has a current object state of S_a and it receives update S_c before S_b, as shown in Figure fig:poc3, then $S_s \neq S_d$ since S_c requires a previous state of $S_s = S_b$ but U_i has a previous state of $S_d = S_a$. This contradicts the assumption that $S_s = S_d$. If U_i had a previous update state S_d other than S_a, say S_x, then update S_a must also be received out of sequence. Again, $S_s \neq S_d$, contradicting the assumption. Similarly, a non-updating state $R_j \in R$ must follow the update sequence set up by the updating states. If R_j has a current update state of S_a and receives S_c before S_b, then again $S_s \neq S_d$ since S_c requires a previous object state of $S_s = S_b$ but R_j has a different previous state S_d since it has not yet received S_b. Since the assumption $S_s = S_d$ is invalid, *P1* must be true for Scenario 2 for any n.

In Scenario 3, we have n sites which independently update the same version S_0 of the object, and $m - n$ sites which also have version S_0 of the object but do not update the object. Let U be the set of updating sites $\{U_1, U_2, ...U_n\}$. Let R be the set of non-updating sites $\{R_1, R_2, ...R_{m-n}\}$. Each $U_k \in U$ updates the object from the consistent initial state of S_0 to S_k. Hence, the set of update states $S = \{S_1, S_2, ...S_n\}$ corresponds to the set of updates generated by each site $U_k \in U$, as shown in Figure 3.7. In other words, each $U_i \in U$ has a current object state of $S_i \in S$, and each S_j expects a previous object state of S_0.

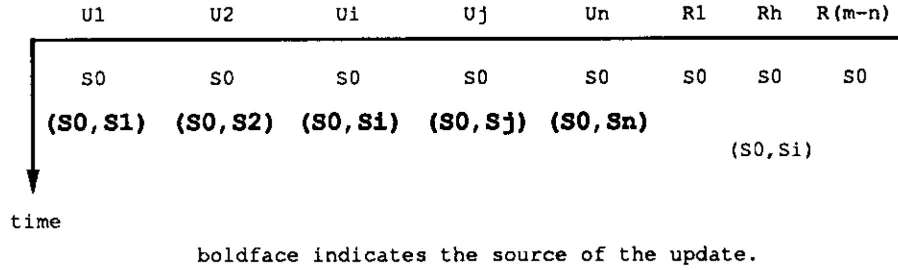


Figure 3.7: Independent Updates with n Updating Sites

When U_k receives update S_l from site U_l, S_l requires a previous object state of $S_s = S_0$. However, since U_k has updated the object from S_0 to S_k, the object available at U_k has state $S_d = S_k$. Therefore, $S_s \neq S_d$, contradicting the assumption that $S_s = S_d$. Similarly, each non-updating site $R_h \in R$ has an initial state of S_0. With the first message it receives from $S_i \in S$ it updates the object from S_0 to S_i. Upon receiving another message $S_j \in S - S_i$, it tries to update the object from S_i to S_j. However, S_j expects a previous update state of $S_s = S_0$ whereas the state of the object at R_h has been updated to $S_d = S_i$, again contradicting the assumption that $S_s = S_d$. Hence, *P1* is valid for Scenario 3.

In Scenario 4, because each update is immediately performed on the source site without network transmission, an updating site $U_i \in U$ will only detect lost messages when it is not updating. Suppose the object is updated from state S_a to S_b, and to S_c, where S_a and S_b are generated by any updating site(s) in U and S_c is generated by any updating site in $U - U_i$. If U_i has a current object state S_a and update S_b is lost, then when S_c is processed at U_i, we would find $S_s \neq S_d$ since S_c requires a previous state of $S_s = S_b$ but U_i has a different state of $S_d = S_a$, as shown in Figure 3.8. This contradicts the assumption that $S_s = S_d$. If U_i has failed to receive S_a as well, then when S_c is processed by U_i, we would again encounter $S_s \neq S_d$ since the object was never updated to S_a and then to S_b. This contradicts the assumption for *P1*. Similarly, a non-updating site $R_h \in R$ must receive all updates in the same sequence as the object was updated. If a non-updating site R_h fails to receive update S_b, then when S_c is processed at R_h, it would find that $S_s \neq S_d$ since $S_s = S_b$ but $S_d = S_a$. Same thing could be said if both messages S_a and S_b were lost, again contradicting the assumption that $S_s = S_d$ and validating *P1* for any n.

P1 can be proven for Scenario 2 even if $x > 1$ update messages are processed out of sequence at site Y, $Y \in U \cup R$. The first update message S_{x+1} that is processed at Site Y will expect a previous object state of $S_s = S_x$ which is not yet available since update S_x (and possibly some messages proceeding it, e.g., S_{x-1}, S_{x-2}) has not yet been processed at Site Y, hence, $S_s \neq S_d$. Similarly, *P1* can be proven for Scenario 4 for any number $x > 1$ of lost updates as long as a later update S_{x+1} is eventually received at Site Y. When S_{x+1} is processed, it expects a previous state $S_s = S_x$ which is not available at Site Y because the update S_x (or some messages proceeding it) has never been processed at Y. Again, $S_s \neq S_d$

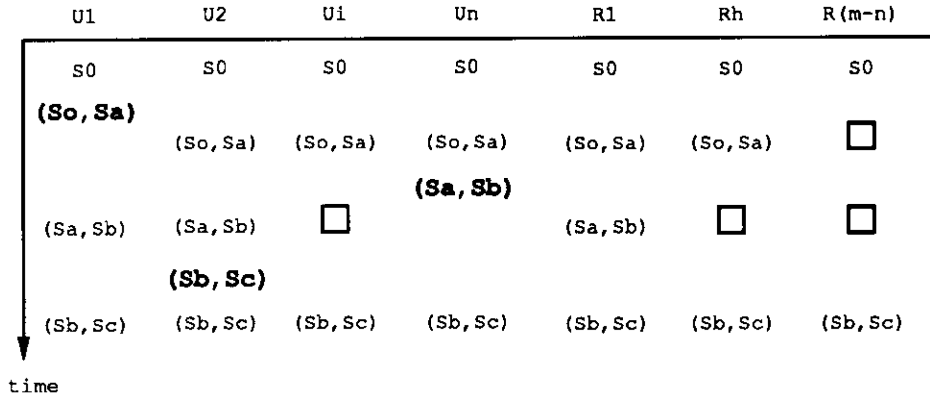


Figure 3.8: Lost Updates with n Updating Sites

and *P1* is proven.

To prove *P2*, we assume that $S_s \neq S_d$ and that there is no conflict. Since Scenario 1 is the only scenario which is conflict-free, we will only consider Scenario 1. Suppose update $S_{j(m)}$ is initiated by Site $U_j in U$ from a previous state of $S_{i(n)}$, then the previous object state of update $S_{j(m)}$ is $S_s = S_{i(n)}$. Since all messages are received correctly and in order, then $S_{i(n)}$ must be processed immediately before $S_{j(m)}$ at all sites Y, where $Y \in U \cup R$, making the previous object state of $S_d = S_{i(n)}$ at Site Y. Again, since all updates must be processed in the same order at all sites, $S_{j(m)}$ must be processed immediately following $S_{i(n)}$. Hence, $S_s = S_d$. This contradicts the assumption that $S_s \neq S_d$ and validates *P2*.

Since both *P1* and *P2* have been proven for any n, then it must be true that *conflict occurs iff* $S_s \neq S_d$. In other words, DDL detects all and nothing but update conflicts.

3.3 DDL with WYSIWIS

As mentioned earlier, conventional locking protocols are overly restrictive for cooperative environments. The nature of write-locks inherently prevents WYSIWIS; updates are not propagated until committed. To solve this problem, we require dirty-read to be supported in coSARA even if the object is locked for exclusive write access. By dirty-read, we refer to the reading of uncommitted data by a transaction other than the lock holder. All updates are immediately broadcast to other sites holding the replica, as if there is no locking. Hence, the same copy of the object will be seen at all sites as it is updated. This requirement has no impact on the operations of DDL which broadcasts all updates as they are performed.

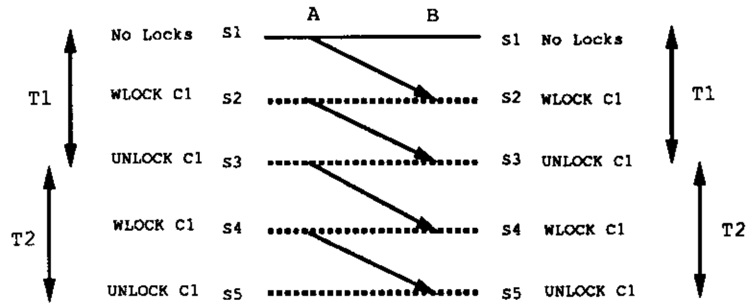


Figure 3.9: Transactions with Dependency Detection Algorithm

However, even in a cooperative setting, a user may wish to work on ideas privately. In coSARA, this can be done through the tools. If the global variable *private* is set to *true*, then the list of existing sites (or sites with a particular object) is masked such that the tool is made to believe that only one site (the local site) exists. Thus, update messages will not be broadcast to other sites. Once the work space is made public (if ever), the site list will be unmasked and made known to the tool, so that all private objects would be broadcast to all other sites in their final private state. To ensure that objects can immediately be broadcast to all other sites, they need to be created with storable-names and storable-ids (as usual) even if they are made while the work space was private. Because this masking and unmasking of site information is done by the tools, it will not affect the operations of DDL. Indeed, all updates will be broadcast to the sites known to the tool, as before.

3.4 DDL Transactions

A transaction is a single execution of a program. It is often made up of a sequence of queries and updates to the database. The goal of transactions is to make a complex operation appear atomic – it either occurs in its entirety, or none at all. For this reason, a transaction is normally considered as both the unit of concurrency and the unit of recovery for database applications [SKAR89]. In non-distributed databases, multi-file consistency can be maintained by using transactions that obey two-phase locking. In a distributed environment, where copies of the object reside on different machines (or even different networks), two-phase locking is no longer sufficient, for the order in which transactions are processed is at the risk of race conditions [ULLM88]. DDL, with its state stamps, ensures all transactions to be executed in the same order at all sites, as illustrated by Figure 3.9. Here, two transactions T1 and T2 are initiated by Site A, both transactions updating only object O1. Because T1 is initiated before T2 at site A, and because transactions are atomic, T2 cannot start execution until T1 is completed. Hence, T2's updates must *depend* on states generated by T1's updates. Since the update requests are broadcast to other sites, this *dependency* dictates all T1's updates to be executed before T2's at all sites. Consider, for example, what

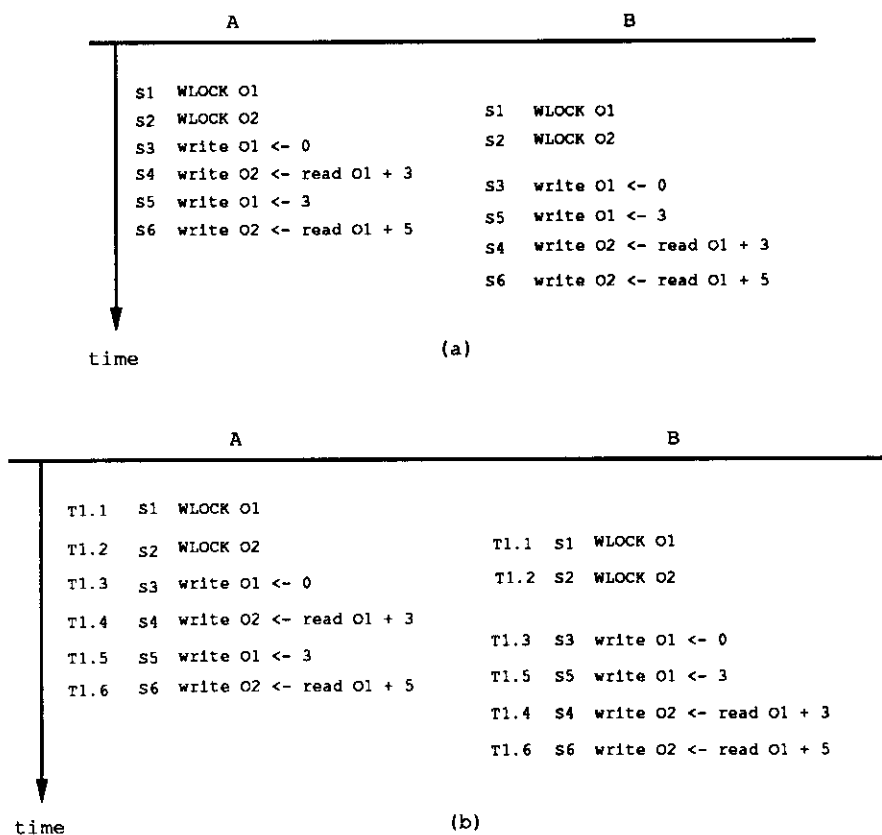


Figure 3.10: Impact of WYSIWIS on Transactions

would happen if, at site B, T2's lock request is processed before T1's updates. T2's request for write-lock depends on a previous O1 state of S3. But because T1's updates have not yet been processed, the current state of B's copy of O1 remains at state S1. We have detected a conflict. Indeed, if transaction Ti is executed before Tj at the originating site, then the state stamps of the object require Ti's updates to be executed before Tj's at all sites. Otherwise, conflicts will be detected due to mismatch of state stamps on the object. This is a secondary effect resulting from the requirement that all updates on an object must be executed in the same order at all sites.

In all previous examples in which only one object is being updated, WYSIWIS is easily supported by DDL by broadcasting all updates as they are processed. Conflicts are detected when updates are processed out of order. This is not sufficient when multiple objects are being updated by a transaction. Consider Figure 3.10a in which only one transaction T1 is active. T1 is initiated by Site A and operates on two objects O1 and O2. Notice that according to A, three updates have been performed on object O1, i.e., S1, S3, and S5, in that order. Three updates have also been performed on O2, those being S2, S4, and S6, in that order. When the update requests are received by Site B, although updates for each object are still performed in the same order as on Site A (S1-S3-S5 for object O1 and S2-S4-S6 for

object O2) the database is inconsistent. Because messages are received out of order at site B, update S4 reads a different value of O1 (O1=3) than what is read by Site A (O1=1) for the same update. This causes the final value of O2 to be inconsistent at the two sites. This conflict would not be detected by the DDL model as described thus far, for read operations are not stamped with previous state of the object, and inconsistencies due to out-of-sequence read-operations are not detected.

This problem is attributed to WYSIWIS because if we could delay the broadcasting of updates until *after transaction commit*, this problem would not have surfaced. But such delay would prevent WYSIWIS which requires all updates to be immediately visible to other sites. A possible solution is to broadcast the *output* of update instead of the *method* of update. However, this solution is undesirable since coSARA outputs are highly graphical and broadcasting such output would mean a lot of encoding and decoding at each site. The solution we propose here is a modified version of DDL. Recall that in DDL, the stamp generated for each update is dependent on the name and the state counter of the originating site. In checking for update conflicts, the previous state of the local copy of the object is compared with the previous state of the same object that is required of the update. We modify this model so that the stamps generated are dependent on the site name and the *transaction counter*. Each update in the transaction is stamped with the name of the site and the value of the transaction counter. At the receiving site, a separate counter is kept to record the value of the transaction stamp of the last update successfully processed. The updates are processed in increasing order of the transaction stamps. Any updates processed out of order will be reported as conflicts. At the originating site, the transaction counter is an attribute of the transaction. It is initialized to 1 as the transaction is created, and is discarded as the transaction terminates. Like the state counter, the transaction counter is incremented with *every* update message generated for that transaction. At the receiving site, a separate transaction counter is maintained for each active transaction. The value of the receiving transaction counter is initialized to 1 upon receipt of the first update message from that transaction. The value is incremented according to the transactions stamp of the last update from that transaction. If the transaction stamp of next message to be processed is not the immediate next integer following the value of the transaction counter, then the updates from that transaction are being processed out of order, hence conflict is detected.

This idea is illustrated in Figure 3.10b. Transaction T1 is initiated by Site A. A transaction stamp is attached to every transaction update. At the receiving site, upon receiving the first update from transaction T1, a transaction counter TC is created for transaction T1 with the initial value of 1. For each transaction update received in order, TC is incremented to identify with the transaction stamp on the last transaction update processed. When the next transaction update to be processed has a different value than what logically follows the value of TC, then conflict is reported. In Figure 3.10b, this occurs when TC expects a value of 4, but the transaction update has a value of 5. Thus, transactions must be processed in the same order as in the originating site.

With this modification, we can still detect all update conflicts as we did with the original DDL (e.g., conflict due to concurrent updates, lost updates, or lock conflicts). The overhead is in maintaining the transaction counters at each receiving site. Assuming N sites, for each transaction generated, N-1 receiving transaction counters are needed, one at each N-1 receiving site. Depending on the number of active transactions from each site, this number may be much larger.

There are several issues regarding coSARA transaction management we must deal with: Should transactions be shared as are other coSARA objects? Should transactions owned by the same user be able to share locks? Should transactions be able to survive session shutdown? How does DDL deal with these issues?

As pointed out earlier, the goal of transactions is to make complex operations appear atomic. It is therefore necessary to require that transactions not be disrupted during its execution. To this end, a transaction should be managed only by one user and one tool. The only time a tool may need to know about the transactions of another tool is during lock negotiation. Even for this purpose, only the transaction ID is needed. Hence, transactions need not be shared among users. This has no impact on the operations of DDL which deals with transactions only in terms of transaction counters used to generate DDL update stamps.

In a case where a user is running two tools, each of which owns a transaction, it is not permissible for the two transactions to share locks (although read-locks are always compatible), because one transaction may corrupt data that the second transaction is using, even though the two transactions are owned by the same user. Because of this restriction, it is possible that these two transactions may be involved in a deadlock. In this case, the user is expected to resolve the deadlock with the aid of the Lock Browser. Again, this requirement has no impact on the operations of DDL which indiscriminately checks for lock conflict based on the lock type, not the locking transaction (with the exception that if a transaction holds the only read-lock on an object, the same transaction can upgrade the read-lock to a write-lock).

The issue with a transaction's ability to survive session shutdown is mainly concerned with uncommitted data and unreleased locks. In other words, if a session is shutdown before the transaction is terminated, should data be committed automatically? Should locks be released? Although it is desirable to have precautionary measures against accidental shutdown, it is undesirable to commit any data without user's consent. It is possible that the user indeed wishes to discard the uncommitted data. On the other hand, any locks left unreleased by the session should be released so that other users can access the objects without mediator intervention.

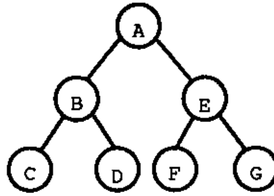


Figure 3.11: Tree Locking and Intention Locking

3.5 Complex Objects

Objects in coSARA are richly hierarchical. It is highly undesirable to allow one transaction to lock the entire complex object, for it significantly degrades sharability. In coSARA, we have chosen to implement the intention-locking protocol which supports the locking of subtrees and uses warning to prevent other transactions from locking the same object [ULLM88]. In this section, we will discuss the pros and cons of this intention-locking protocol, and we will discuss how DDL can be implemented in this protocol.

The most rudimentary locking protocol for hierarchically structured objects is the "tree-locking." Except for the first object to be locked in the tree structure, no object can be locked unless the same transaction has a lock on the parent. Once an object has been locked, the lock on the parent can be released. In this protocol, a lock on an item implies a lock on all of its descendents. An example clarifies this concept. Consider the hierarchically structured objects in Figure 3.11. Transaction T1 first locks object B. By locking B, it implicitly possesses the lock on all of B's descendents. If it is later determined that T1 no longer needs to lock all of B's descendents, but it only needs C, then T1 can lock C and release its lock on B. This way, other transactions can lock other descendents of B, increasing sharability. A major problem with the tree-locking protocol is that an object may be locked twice by different transactions. Consider our example in Figure 3.11 again. Transaction T1 has a lock on B and transaction T2 has a lock on A. Since a lock on an object implies a lock on all of its descendents, B (and all its descendents) are actually locked twice, by both transactions T1 and T2.

To avoid this conflict, the intention-locking protocol [ULLM88] requires that the transaction place a "warning" on all its ancestors before locking the object itself. A warning on an item prevents other transactions from locking the object but it does not prevent other transactions from also placing a warning on the object or from locking some descendent of A that does not have a warning. A transaction that obeys the intention-locking protocol must first place a lock or warning at the root. It proceeds down the tree until it reaches the object it wishes to lock. The transaction cannot place a lock or warning on an item unless it holds a warning on its parent. When the transaction is ready to commit, it releases the lock on the object and warnings or locks on its ancestors. The protocol requires that no transaction can remove a lock or warning on an object before it releases all locks and warnings on all

of its descendents. Finally, to ensure serializability, the protocol requires that all unlocks follow all warnings and locks. Again, let's clarify this with an example. To update B, T1 must first place a warning on A, then lock B. Another transaction T2 can update E by also placing a warning on A and then locking E. The two warnings do not conflict since warnings are compatible. Because locks and warnings *do* conflict, no transaction can lock A while T1 and T2 have warnings on A. Since all locks must start with warnings on the ancestors, no object in this protocol can be locked twice.

To support such data-sharing in both the locking and non-locking modes in coSARA, we require that an object can be updated if and only if the updating transaction has a write lock (explicit or implied from a locked ancestor) on the object *or* no transaction has a lock or warning on the object and no transaction has a lock on any of its ancestors. To see how this works, consider the following scenario (again using Figure 3.11). Transaction T1 has warnings on objects A and B, and it has a write lock on D. Since T1 is the only one to have a write lock on D, it is the only transaction that can update D. On the other hand, since no transaction has a lock on C or any of its ancestors (although T1 has warnings on A and B), any transaction can update C. If a second transaction T2 has a warning on A and a read-lock on E, then no transaction, including T2, can update E or any of its descendents until either T2 releases its read-lock or upgrades it to a write-lock. In the former case, any transaction can update E, F, and G. In the latter case, only transaction T2 can update E and its descendents. The reason that updates are not allowed on an object with warnings is the same as that for disallowing locking of an object with warnings it may modify the semantics of the relationship between the parent object and any locked descendents. For example, while transaction T1 has warnings on A and B and a lock on D, it is highly undesirable for another transaction T3 to, say, delete the connection from B to D, or even from A to B. Hence, no update should be performed on objects with warnings.

Warnings should be implemented in the same fashion as lock requests and releases in DDL in the sense that it should be treated as an object update, with a new DDL update stamp on the object. In fact, the warnings can be stored in the same attribute as locks on the object. A valid question at this point is whether we can represent warnings as read-locks internally. After all, read-locks are compatible with read-locks, just as warnings are compatible with warnings. For DDL, the answer is no. Read-locks are semantically different from warnings in the sense that a read-lock on an object implies a read-lock on each of the object's descendents. Hence, logically no other transaction should be able to update an object (or its descendents) with a read-lock. However, if we were to represent warnings as read-locks, then no scenario would satisfy the second condition (defined in the last paragraph) to update an object. In fact, if we were to represent warnings as read-locks, then for a transaction T1 to acquire a write-lock on D in Figure 3.11, T1 would effectively have to have read-locks on A and B. Say transaction T4 now wants to update C. It cannot do so because it doesn't have a write-lock on C (i.e., it cannot satisfy the first condition), and there is a lock on B (i.e., it can't satisfy the second condition). In other words, to update on object, a transaction *must* have a write-lock on it. This fails to meet the requirement that

DDL must support data-sharing regardless of presence of locks. Hence, warnings *must not* be treated as read-locks in DDL.

Consideration has been given to the possibility of providing both tree-locking and intention-locking protocols in coSARA. However, this is undesirable for two reasons: First, as pointed out earlier, tree-locking does not prevent two transactions from effectively locking the same object. Second, in the presence of both protocols, lock enforcement can be quite difficult since the lock manager would have to recognize two semantically different locking schemes.

3.6 Recoverability and Transaction Abort

In the current implementation of DDL, objects are saved when a transaction is ended, when the collaborative session is terminated, or upon user demand. This is by no means sufficient, especially if we wish to allow transaction abort or support failure recovery. A very important issue regarding transaction abort in a collaborative environment such as coSARA is the duration of transactions. In the case of coSARA, transactions may last an arbitrarily long time, up to the duration of the eight-hour collaborative session. It is therefore undesirable for a user to mistakenly abort a transaction that may have lasted several hours. Even though up-to-date replicas may have once existed on other sites during the collaborative session, these replicas would also be undone as the transaction aborted. Failure recoverability is important for the same reason, with the added disadvantage that updates not made by transactions may never have been saved.

Two of the most common protocols that guarantee recoverability and avoid cascading rollback are strict two-phase locking and strict timestamp-based concurrency control [ULLM88]. Both protocols perform all updates in the workspace. A transaction cannot write into the database until it has reached its commit point. Thus, cascading rollback is prevented since reading of uncommitted data is not permitted. Both protocols keep a "log" of all the changes made to the database and the status of each transaction. After the transaction has committed, a record is written into the log, which is copied to stable storage; and then the value is written into the database itself. When a system failure occurs, a re-do algorithm is executed which examines the log and restores the database. Although both protocols are quite reasonable for conventional databases, they cannot fulfill coSARA's needs. For one thing, it requires all the updates to have been made by transactions. This is not always appropriate in the collaborative context which does *not* require the use of locks or transactions. Second, because dirty-read is disallowed, WYSIWIS cannot be supported by these protocols. Because of these two special needs of the coSARA database, other protocols must be sought to support transaction abort and aid in recoverability.

Recoverability is easier to achieve than to support transaction abort. First, to recover from a multi-site collaborative session is easy if we assume independent site failures. The

failed site can simply obtain an up-to-date copy of the database from a peer collaborative session. All locks held by transactions from the failed session at the time of the crash must be released by the system so that other sites in the session can access the objects before the failed site recovers. Users must negotiate to determine if uncommitted transactions should be committed.

To recover from a single-site session is a little more complex since the database may not have an up-to-date copy of the objects and no other site holds a replica of the objects. A reliable but expensive method is to have periodic snapshots of the system which are taken independent of session and transaction status. The snapshots don't need to be of the entire coSARA object-world, just of the objects modified since the last save. This is easily accomplished since we can simply have a "dirty-bit" on every object and take snapshots only of the objects whose "dirty-bit" has been set. If the time interval between periodic snapshots is short, this scheme ensures recoverability with or without locks and transactions.

To support transaction abort is complex since coSARA's requirement of WYSIWIS necessitates dirty-read and often leads to cascading rollback. A simple solution – by no means optimal – is to take advantage of user collaboration and take periodic snapshots of the object-world, independent of session and transaction status. When a transaction is aborted, users can then negotiate on the rollback point. They can choose to retreat to the object-world state provided by a snapshot, or they can retreat to the previous commit-point. This scheme requires frequent snapshots and hence may be memory expensive. Since not all objects may be stored at all sites (recall that coSARA supports *replication on demand*), this process may have to be done by the centralized object-world database server. The server can obtain a copy of the global-directory, gets a copy of each object from a storing site, and saves a snapshot of each object in the database. Fortunately, because the role of the centralized database server is minimal, this process should not cause a big CPU overhead. Earlier snapshots can be deleted if users agree that the snapshot is obsolete. At the end of the collaborative session, these snapshots can be purged since all transactions must be terminated and we no longer need to be concerned about transaction abort or cascading rollbacks.

3.7 DDL Optimization

As pointed out in Sections 3.1.2 and 3.1.3, there are times when reporting update conflict may not be necessary, particularly if intermediate updates are not needed by the receiving site or if lost messages are those generated for a locked object by the lock holder. Unfortunately, this is not as simple as it may seem. Consider the example in Figure 3.2c. Although it is possible for DDL to detect that read-locks are compatible and to proceed to grant the locks to both sites without reporting lock conflicts, the problem is how to assign a consistent update stamp without disrupting DDL. (Recall that if no conflicts are reported, then DDL guarantees that eventually all sites will agree.) If Site A assumes B's update stamp (S3) while

B assumes A's update stamp (S2), then both sites remain inconsistent. If DDL arbitrarily assigns a new value for the update stamp, how do we make sure all copies of the object receive this new stamp? We can certainly force all sites to assume the stamp of a previous consistent state, but this is difficult to determine since intermediate update messages may have been lost.

We face a similar problem if we consider the example in Figure 3.3c. In this case, if B could recognize the fact that Site A is the only site updating the object (e.g., A has the write-lock), and B does not need any intermediate values of A's updates, then it would be more efficient if B simply assumed A's updates as accurate. The flaw is that a third site may also be making updates to the object. B has no way of knowing which of the updates is accurate. If B simply assumes the values from another site, A or C, although no conflicts will be reported by B, conflicts are sure to surface later on another site. Hence, by having B assume another site's value, conflicts will be detected later, when it is more difficult to resolve.

One optimization we *can* do is to set up message buffers at each site. With each update message received, DDL first checks to see if the message can be processed immediately without causing a conflict. If so, the message is immediately processed. If not, we put the message on a timed queue. As new update message X arrives, if the message is processed without causing a conflict, DDL tries to process other messages that were delayed (and hence put on the queue) because X was not yet processed (i.e., updates that *depended* on X). Periodically, the messages on the queue will be checked for *time-out*. If the update message has been on the queue longer than the allowed *time-out*, then either messages have been lost or concurrent updates took place, hence conflict is reported. With this scheme, scenarios as shown in Figure 3.10b would not cause conflict at Site B if updates T1.4 arrives before time-out expires for update T1.5.

3.8 Overhead of DDL

The time it takes in resolving conflicts is a major overhead of DDL. Most of this overhead is incurred when users are working on different networks connected by bridges and gateways. When users are working on the same network, it is very likely that all update messages will be received in the same order at all sites, hence no conflicts would be reported. However, when users are working on interconnected networks, every message processed out of order requires user intervention. With the aid of message buffers, this may be made less cumbersome.

The time overhead in negotiating for locks can be reduced in coSARA by using a lock browser which displays all current locks and their lock owners. When too much time is spent on negotiating object access, participants should reconsider the division of labor within the group in an attempt to reduce the number of conflicts.

Storage overhead in DDL is incurred: by storing the value of the previous update stamp and the lock information with every object; by storing the transaction counter as an attribute of transactions; by maintaining one transaction counter per site per active transaction; and most of all, by maintaining a message buffer at every site.

Network overhead in DDL is caused by broadcasting updates before the update is committed. This is a necessary cost since we want to support WYSIWIS.

CHAPTER 4

DDL and Remote Sharing: The Implementation

To appreciate the implementation of the coSARA database – the "object-world" – and the remote sharing capabilities, one must first understand the underlying properties of coSARA objects and the operations performed on these objects. As mentioned earlier, data in the coSARA object-world exists as objects, and as such, they have prescribed inheritance and encapsulation properties [DITT86, DIED89, NIER89]. In this chapter, we will first describe how object-world objects and operations behave. We will then describe how DDL update conflicts are detected, how locking is enforced, and how conflicts are resolved. Finally, we will illustrate how remote data sharing is supported.

4.1 Properties of Objects

To ensure that all coSARA objects can be shared across sites and stored in the persistent database, each coSARA object has a unique and immutable identifier. To be exact, this identifier is unique across all sites, at all times, and is immutable regardless of the changes made on the object. This identifier is used only for internal references. To the user, each object is identified by its name and type. Therefore, to ensure that each object can be correctly accessed at any time and from any site, there is a one-to-one mapping between each object and its identifier. This is done by defining all coSARA objects to be of class or subclass *transmittable*:

```
(defclass transmittable ()
   ((storable-id   :type string
                   :initarg :storable-id
                   :reader storable-id)
   (storable-name   :type string
                    :initarg :storable-name
                    :reader storable-name)))
```

All objects of this *transmittable* class has two attributes – the "storable-id" and the "storable-name." The former contains the unique and immutable identifier and is generated by the object-world function call

(unique-sym).

The latter contains the name of the object as specified by the user. In short, each *transmittable* object is created with a system-generated identifier and a user-specified object name. Consider, for example an object class *counter* which is defined by

```
(defclass counter (transmittable)
  ((value :accessor value :reader value :initform 0)))
```

The *counter* class has one attribute *value* which contains the current value of a counter object. This attribute has an initial value of zero and its value can be accessed by "value." By making it a subclass of *transmittable*, it inherits the "storable-id" and "storable-name" properties. Note that the only part of this definition that is related to the object-world is the superclass *transmittable*. There is multiple inheritance in coSARA, so that the *transmittable* class may be one of several superclasses from which the *counter* class inherits its properties and behaviors. When an object of class *counter* is created by

```
(setf c1 (make-instance 'counter :storable-name "counter1"))
```

the variable *c1* points to an object of type COUNTER, initialized with storable-id "RA.surya-123456," storable-name "counter1," and the counter value 0. Note that the storable-id is generated by the coSARA object-world automatically, and the storable-name receives the user-specified value.

However, this is not sufficient for objects to be shared and stored. Each object must have an ASCII representation by which it is transmitted between sharing sites and saved in a persistent database. This ASCII representation contains the object class, its storable-name, storable-id, all attributes of the object and the associated attribute values. This ASCII representation is generated by calling function *send-object* with the object. For example, by calling function *send-object* with object *c1*i, we obtain *c1's* ASCII representation

```
"(COUNTER \"c1\" \"RA.asa-105558\" (VALUE . 0))"
```

where the fourth element of this string contains the counter attribute VALUE and its current value of 0. It is this ASCII representation of the object that is transmittable across sharing sites and stored in the database. Upon receipt of the object's ASCII representation, whether from another site or from the database, each site parses the string, interpreting the first element of the string as the object type, the second element as the object storable-name, the third element as its object storable-id, and all other elements as pairs associating each

attribute with its attribute value. Therefore, for an object to be transmitted from one site to another, we first encode it in its ASCII representation by calling function *send-object* at the sending site. Then at the receiving site(s), this ASCII representation is decoded accordingly, with a replica of the object created based on the type and attributes of the object.

Let's consider a more complex example in which data objects are hierarchically structured. We have the following class definition for a binary search tree:

```
(defclass btree (transmittable)
   ((key :accessor key     :initform 0)
   (left :accessor left    :initform nil)
   (right :accessor right    :initform nil)))
```

An object of this class has three attributes (besides those inherited from its superclass *transmittable*), one for the key value of the node, one for its left subtree, and a third for the right subtree. We can create an object of this class, assigning to it storable-name "root" and key value 15:

```
(setf root (make-instance 'btree  :storable-name "root"))
(setf (key root) 15)
```

Again, we can get the ASCII representation of this object by calling function *send-object*. We get

```
"(BTREE  \"root\"
         \"RA.asa-105558\"
         (KEY . 15)
         (LEFT . NIL)
         (RIGHT . NIL))"
```

Again, the first element of the string is the class of the object. The storable-name and storable-id follow as the second and third element of the string. The storable-name is specified by the user while the storable-id is generated by the object-world (in this case, we assume a value of "RA.asa-105558.") The remaining three elements are (attribute, value) pairs, one for each of the three attributes of class *btree*. Note here that function *send-object* is smart enough not to encode the attributes class *btree* inherited from class *transmittable*. After all, these attributes are already encoded as the second and third elements in the ASCII representation.

A pointer to a *transmittable* object is encoded as a call to function *read-object* with the storable-id of the referenced object as its argument. What *read-object* does is that it uses the given storable-id to locate a replica of the object in the local database and returns a pointer reference to the local replica. Since storable-id is a unique and immutable property of the object, it guarantees to return the correct replica. If it fails to find a replica of the object from the local database, it requests an ASCII representation of the object from a site which does hold a replica. When that also fails, it reads it from the persistent database. An example will clarify this. We first create two more nodes in this binary search tree:

```
(setf node-1 (make-instance 'btree  :storable-name "left"))
(setf node-2 (make-instance 'btree  :storable-name "right"))
(setf (key node-1) 10)
(setf (key node-2) 20)
(setf (left root) node-1)
(setf (right root) node-2)
```

Thus, the left subtree contains a search tree with key value 10, while the right subtree contains a search tree with key value 20. Neither of these subtrees contain pointers to other subtrees. When object *root* is encoded by *send-object*, we get

```
"(BTREE \"root\"
        \"RA.asa-105558\"
        (KEY . 15 )
        (LEFT . (read-object  :id \"RA.asa-105560\"))
        (RIGHT . (read-object  :id \"RA.asa-105561\")))"
```

where string "RA.asa-105558" and similar ones are the unique identifiers of the objects. So the ASCII representation of the root indicates that the key is (still) 15, but the left and right subtrees now have a value which is a function call to read-in the correct subtrees. At the receiving site (or when the object is read-in from the persistent database), the *construct* function will be called to construct the object from its ASCII representation. When it encounters an (attribute, value) pair where the value is a call to *read-object*, it evaluates the function call with the given storable-id and fill in the proper value for the left and right subtrees. This will occur recursively, so if node-1 and node-2 have their own subtrees, these will all be read in all the way down to the leaves.

The reason that we encode the pointer as a function call to *read-object* instead of the ASCII representation of the object being pointed to is an important one: We should always use the local replica of the object whenever possible. This can be achieved by calling *read-object*. If we instead encode the pointer with the ASCII representation of the object being pointed to, then a new replica of the object is created even though a replica may already

exist at that site. Worse yet, pointers that logically point to the same object may not do so physically. An example clarifies this concept. Consider the following class definition and sequence of operations.

```
(defclass person (transmittable)
   ((child   :accessor child   :initform nil)))

(setf baby  (make-instance  'person   :storable-name "baby"))
(setf dad  (make-instance  'person   :storable-name "daddy"))
(setf mom  (make-instance  'person   :storable-name "mommy"))
(setf (child dad) baby)
(setf (child mom) baby)
```

Note that *person* objects *mom* and *dad* point to the same child object *baby*. Encoding them properly, we would have

```
"(PERSON  \"baby\"  \"RA.asa-105562\"  (CHILD . NIL))"
"(PERSON  \"daddy\"
          \"RA.asa-105563\"
          (CHILD . (read-object :id  \"RA.asa-105562\")))"
"(PERSON  \"mommy\"
          \"RA.asa-105563\"
          (CHILD . (read-object :id  \"RA.asa-105562\")))"
```

Suppose that a second site which does not hold a replica of objects *baby, dad,* and *mom* now requests a replica of *dad.* It gets the ASCII representation for *dad*, creates a new *person* object, assign the new object the same storable-name and storable-id as that in the ASCII representation of *dad* it received from the first site, and evaluates the call to *read-object* for its *child* attribute. Since the second site does not have a replica of object "RA.asa-105562," function *read-object* proceeds to obtain a replica from the first site. This request is responded by the first site with the ASCII representation for *baby.* Upon receiving the ASCII representation for *baby*, the second site creates a new *person* object and assigns it the same storable-name and storable-id as that in the ASCII representation of *baby.* The *child* attribute of *baby* is set to NIL, just as dictated by the ASCII representation. Now the second site has a replica of *dad* and a replica of *baby*, with the *child* of *dad* pointing to *baby.* Suppose now the second site requests a replica of *mom.* It receives the ASCII representation for *mom* and once again evaluates the call to *read-object* to fill in the *child* attribute. This time, since there is a local replica of object "RA.asa-105562," *read-object* returns a pointer to the local replica. Therefore, replicas of *mom* and *dad* on the second site also point to the same *baby* object. Any changes made to the *child* of *dad* would be reflected in the *child* of *mom* since they refer to the same *child.*

If, instead of encoding the pointer to object *baby* as a call to *read-object*, we encode it with the ASCII representation of *baby*, then two replicas of *baby* will be created at the second site, one from reading in *dad*, and a second from reading in *mom*. In fact, *dad* and *mom* would be pointing to different objects, although both objects have the same attributes. This means that changes made to the *child* of *dad* would not be reflected in the *child* of *mom*. This changed the semantics of the operations. Further, the fact that these two objects share the same storable-id violates the requirement for one-to-one mapping between each object and its identifier. Therefore, pointers must be encoded as calls to *read-object* to ensure correctness of the operations.

The behavior we have described thus far for objects in the object-world is precisely the behavior we exploit to create, encode, and store DDL objects. The *DDL* class is defined as

```
(defclass DDL  (transmittable)
   ((ddl-stamp   :accessor ddl-stamp   :initform nil)
   (ddl-locks   :accessor ddl-locks   :initform nil)))
```

In other words, all objects that are of class or subclass *DDL* will inherit attributes "storable-id" and "storable-name" from the *transmittable* class and will have attributes "ddl-stamp" and "ddl-locks" from the definition of class *DDL*. Attribute "ddl-stamp" is used to record the stamp of the last update performed on the object. Attribute "ddl-locks," on the other hand, keeps track of all read/write locks placed on the object as well as warnings from tree-lock (as discussed in Section 3.5). The initial value of attribute "ddl-stamp" is NIL to indicate the fact that there has yet been no update on the object. The initial value of attribute "ddl-locks" is also NIL since all DDL objects are created with no locks. For example, if we were to define a new class *ddl-counter* which is a subclass of *DDL*

```
(defclass ddl-counter (ddl)
   ((value   :accessor value   :reader value   :initform 0)))
```

and create a new object c2 of class *ddl-counter* by

```
(setf c2   (make-instance 'ddl-counter   :storable-name "counter2"))
```

Then object c2 would have five attributes – storable-id, storable-name, ddl-stamp, ddl-lock, and value – with attribute values "RA.surya-123457," "counter2," NIL, NIL, and 0, respectively. Its ASCII representation appears as

```
"(DDL-COUNTER \"counter2\"
```

```
\"RA.surya-123457\"
(DDL-STAMP . NIL)
(DDL-LOCKS . NIL)
(VALUE . 0))"
```

It is worth emphasizing here that the only difference between a class definition in DDL and a class definition in the Common Lisp Object Systems (CLOS) is the reference to the superclass *DDL*. Hence, a user who is familiar with CLOS can take advantage of the object-world functionalities without much effort.

Before leaving this section, we must discuss how each site manages its local database. Each site has two directories: the "global-directory" and the "local-directory." The "global-directory" has an entry for each object in the collaborative session and keeps track of the sites that contain a replica of the object. The "local-directory," on the other hand, has an entry for each object for which the local site holds a replica. The entry contains the storable-id (or name and type) of the object, and a pointer to the object. The relationship between these two directories is as follows: When an object is referenced, the local site first looks for it in the local-directory. If it finds it in the local-directory, the local replica is used. Otherwise, it looks at the entry for the object in the global-directory and gets the list of sites which *do* hold a replica of the object. It then requests a replica of the object from one of these sites. If it fails to find an entry for the object in the global-directory as well, then it searches for the object in the persistent database.

When an object is created or loaded from the persistent database, the local site enters the object identifier into the local directory. If the object has a name, the name and class are also entered. This information is updated in the global-directory by inserting a new entry in the global-directory which contains the identifier (and name and type, if any) of the object, and indicate that the local site has a replica (and the only replica) of the object.

It is not sufficient to update this information in the local site. We must inform all other sites that the local site has a replica of this object. This is achieved by making the global-directory a *transmittable* object. Therefore, when the local site updates its global-directory, as described above, all remote sites receive a message so that they also update their global-directories. In the future, if any of the remote sites needs a copy of the object, it will look it up in its global-directory, and obtain a replica from a site currently holding the replica.

If a site requests an object from a remote machine, only the local-directory entries have to be created. The global-directory site list for this object has to be updated to include this new site.

```
(defclass counter ()
  ((value :accessor value :initform 0)))

(defmethod increment ((c counter))
  (setf (value c) (+ 1 (value c)))
  c)
```

Figure 4.1: CLOS Definition for Class Counter and Method Increment

```
(defclass counter (ddl)
  ((value :accessor value :initform 0)))

(defbroadcast increment ((c counter))
  (setf (value c) (+ 1 (value c)))
  c)
```

Figure 4.2: Method Increment Which Updates All Replicas

4.2 Properties of Operations

Now that we understand how coSARA objects are stored and encoded, we must illustrate how operations on these objects are performed, and how updates are broadcast to other sharing sites in order to maintain a consistent view of the database across all sites. Let us begin by looking at how functions are defined in CLOS. Lines 1 and 2 in Figure 4.1 contains the CLOS class definition for class *counter*, as we have seen earlier. The *increment* method in Figure 4.1 is a function defined on all objects of class *counter*. It takes as argument an instance of the *counter* class and adds one to the current value of the counter instance. The return value of the function is the *counter* argument itself. Suppose we have an instance of class *counter*, let's call it *"c1,"* whose current value is zero. By applying

(increment (increment c1)),

the value of *c1* is incremented to two.

But merely incrementing the value of the *counter* object at the site which issued the *increment* request is not sufficient for the coSARA object-world. All replicas of the object should be incremented to preserve database consistency, regardless of the location of the replica. The definition of the *increment* function which achieves this "global update" is shown in Figure 4.2. Notice that the only difference between the definition of *increment* in Figure 4.1 and that in Figure 4.2 are the words "defmethod" and "defbroadcast." This simplicity is the result of a special feature of the *transmittable* class – the *defbroadcast* macro. Functions defined with *defbroadcast* operate on all replicas of a *transmittable* object,

regardless of where the replica is located. Since class *counter* in Figure 4.2 is a subclass of *ddl* and since *ddl* is a subclass of *transmittable*, all instances of *counter* are *transmittable* objects.

When the code written with *defbroadcast* is macro-expanded, three functions are generated. The first function contains the code that implements the required functionality at the source site (the site which issued the function call). The second function generated by the *defbroadcast* macro is an ":after" function to the first function. In CLOS, each ":after" function is associated with a primary function and is executed immediately after the primary function runs to completion. Since the first function is run at the source site, so is this :after function. The goal of the :after function is to inform other sites of the change in object state. It does so by transmitting to other sites a TCP/IP datagram message which contains the function call and argument(s) to the function call. The function executed at the remote sites is the third function generated by the *defbroadcast* macro. It is a plain function that implements the required functionalities of the original code. The reason that we differentiate between the first function and the third function is because the third function does *not* have the side effect of broadcasting the function call as does the first function. In other words, the :after function generated by *defbroadcast* is only executed at the source site when the first function runs to completion. It is not executed at any of the remote sites. If the :after function were to run on the remote sites as well, then the system would go into an infinite broadcast loop.

There is another problem that can cause extra (though not infinite) messages to be broadcast. This happens when one broadcast function calls another broadcast function. The source site must be smart enough to announce only the first function call and not the second function call. Otherwise, all remote sites will perform the second operation twice. Further, upon receiving the first broadcast message, a remote site executes the "non-broadcast" version of the function, which contains a call to the nested broadcast function. The second broadcast function has to be smart enough not to broadcast to other sites since other sites are independently executing that function as a result of the first function. This problem is dealt with by having a dynamically scoped variable named *sync* which is set to true if a broadcast function is currently being executed. The dynamic scoping means that the variable can be referenced anywhere as long as its binding is currently in effect.

Another problem is that a single function call may require several arguments. but not all arguments exist at all sites. If a function of this nature is going to be executed, a remote site may try to apply the "non-broadcast" function to some objects that do not exist on that site. Function *send-args* solves this problem by making sure all arguments to the function call have the same site list. If a site does not have a replica of each argument to the function call, a replica of the object is sent.

To understand all of this better, the macro-expanded code for *increment* is shown below. Figure 4.3 contains the function to be executed at the source machine. Figure 4.4 contains

```
(defmethod increment ((c counter))
  (let* ((args (list c))
         (local-stamps (cons 'list (mapcar #'get-ddl-stamp args))))
    (if *sync*
        (progn
          (setf (value c) (+ 1 (value c)))
          (setf ret-val c))
        (let ((*sync* nil))
          (declare (special *sync*))
          (setf *sync* t)                                              10
          (send-args (flatten-out (list (get-id c))))
          (setf (value c) (+ 1 (value c)))
          (setf ret-val c)))
    (setf *ddl-stamp-before* local-stamps)
    (setf *ddl-stamp-after* (cons 'list (mapcar #'get-ddl-stamp args)))
    ret-val))
```

Figure 4.3: The Broadcast Version of Increment

the :after function which is to be executed at the source machine immediately after the primary *increment* runs to completion. Finally, Figure 4.5 contains the version of *increment* to be executed on the "remote machine" when the message generated by the :after function in Figure 4.4 is received.

When *increment* is called with *counter* object *c1*, the function shown in Figure 4.3 is called at the source site. First, the DDL stamps on the arguments are recorded (as shown on Line 3). In this case, *increment* has only one argument, the counter. Then it evaluates the dynamic scoping variable *sync* to determine if it is already within the scope of a broadcast function. If so, the statements of the original *increment* function are executed (Lines 6 and 7). Otherwise, variable *sync* is set to true before the statements are executed (Lines 8 to 13). After all the statements have been executed, the values of the DDL stamps *before* and *after* the update are assigned to *global* variables *ddl-stamp-before* and *ddl-stamp-after* for latter references (Lines 14 and 15).

Several things are worth noting here. First, the original *increment* function as shown in Figure 4.2 returns a pointer to the *counter* argument to the calling function. To preserve the semantics of the code, the value returned by the last statement of the *defbroadcast* function (as shown in Figure 4.2) is assigned to the local variable "ret-val" (Lines 7 and 15). Then at the end of the macro-expanded *increment*, this variable is used for the return value (Line 18). Hence, the semantics of the *increment* function is preserved.

Second, Lines 8 to 13 are executed only if the dynamic scoping variable *sync* is false at the start of the function call. Recall that *sync* is true if the function is within the scope of another broadcast function. We know that if Line 8 is executed, it must mean that we

```
(defmethod increment :after ((c counter))
  (when (not *sync*)
        (with-scheduling-inhibited
         (announce-update
          (ow-update-socket *ow*)
          (format nil
                  "(propagate-update ~s ~s~s)"
                  (package-name *package*)
                  (get-dests (flatten-out (list (get-id c))))
                  (cons 'nb-increment                                    10
                        (append (list *ddl-stamp-before*
                                      *ddl-stamp-after*
                                      *current-transaction*
                                      (storable-name *session*))
                                (list (encode-all c)))))))))
```

Figure 4.4: The :after Method of Increment

```
(defmethod nb-increment (ddl-before ddl-after transaction session (c counter))
  (let* ((args (list c))
         (local-ddl-stamp (mapcar #'get-ddl-stamp args)))
    (if (equal ddl-before local-ddl-stamp)
        (let ((*current-transaction* nil))
          (declare (special *current-transaction*))
          (setf *current-transaction* transaction)
          (if *sync*
              (progn (setf (value c) (+ 1 (value c)))
                     (setf ret-val c))                                   10
              (let ((*sync* nil))
                (declare (special *sync*))
                (setf *sync* t)
                (setf (value c) (+ 1 (value c)))
                (setf ret-val c)))
          (do ((x args (cdr x)) (y ddl-after (cdr y)))
              ((null x))
              (when (typep (car x) 'ddl) (setf (ddl-stamp (car x)) (car y))))
          ret-val)
        (error "Update Conflict Detected~%"))))                          20
```

Figure 4.5: The Non-broadcast Version of Increment

were not originally in the scope of a broadcast. But since the *increment* function is defined with *defbroadcast*, and we are currently in the scope of *increment*, then we must have just entered a broadcast scope. Therefore, we must mark this fact by setting *sync* to true (Line 10) so that other broadcast functions that may be called in this scope would no longer be broadcast. Further, Line 11 calls *send-object* to ensure all arguments to the function call have the same site list. If a site does not have a replica of the object, a replica is sent.

In Figure 4.4, we have the code to be executed on the source site after the primary *increment* function (Figure 4.3) runs to completion. We first check to make sure that we are not in the scope of a broadcast. Because the :after function is not inside the scope of the primary function, the *sync* variable here has the value of the calling function (*not* the primary *increment* function). If the calling function were a broadcast, then *sync* on Line 2 would be evaluated to true. On the other hand, if the calling function weren't within the scope of broadcast, *sync* would be evaluated to false. If it is the case that *sync* is false, then we must broadcast the function call so that all replicas of the object would also be updated. Indeed, that is exactly what we do on Lines 2 to 15. Here, a message is generated to announce the function call, and this message is sent by function *announce-update* through the designated "update-socket." This message (as shown on Lines 7 to 15) contains a call to function *propagate-update*, with the function package name, the list of sites who should process the message, and a function call. This function call is encoded in string with the name of the non-broadcast function to be called at the remote site ("nb-increment"), the DDL stamp of the counter object *before* the update, the DDL stamp of the counter object *after* the update, the transaction which requested the update, the session which owns the updating transmission, and finally the ASCII representation of the counter object that was used as argument in the call to *increment*. The sites which are expected to process the function call are those which store replicas of the counter argument, as shown on line 9. This message is sent by function *announce-update* to all sites in the collaborative session.

Notice how the primary function and the :after function work together. In the primary function, if it is determined that the function call must be announced to other sites, *send-object* is called to ensure all arguments to the function call have the same site list. Then in the :after function, we assume that all sites which will execute the function call have a replica of all the arguments it needs, and simply announce the function call to the appropriate sites, with no checking done. In addition, the values of the DDL stamp are put into the global variables *ddl-stamp-before* and *ddl-stamp-after* by the primary-function and are retrieved from these global variables by the :after function, which later sends the information to other sites for consistency check.

The reason that the function to be executed at the remote sites has a different name ("nb-increment" instead of "increment") is because function *increment* has an :after function which is automatically executed and which causes the function call to be broadcast. We do not want the remote sites to broadcast the function call as well. Hence, a different function name is used.

```
(defddl counter ()
  ((value  :accessor value  :reader value  :initform 0)))

(defmethod (setf value) :before (value (object counter))
  (if (confirm-lock object *current-transaction* :write)
      (setf (ddl-stamp object) (unique-sym))
      (error "Update Request Denied")))
```

Figure 4.6: DDL Locking Enforcement

Function *nb-increment* in Figure 4.5 is simpler. Each site on receiving a propagate-update message, will check the site list. If it is on the site list, it executes the function call. In this case, the function call is *nb-increment*. Within the function *nb-increment*, the DDL stamps on the local replicas of the arguments are compared with those on the source site before the update (Line 4). If the stamps match, the statements of the original *increment* are executed, with the final stamps on the local replica of the argument set to those indicated in the update message (Lines 8 to 19). On the stamps fail to match, then update conflict is signaled.

4.3 DDL Locking Enforcement

Enforcement of locks in DDL are done automatically with the *defddl* macro, as shown in Figure 4.6. Let's use our *counter* example again. Instead of defining the counter class with *defclass*, a new macro *defddl* is defined. When a class is created with *defddl*, it automatically becomes a subclass of *ddl* (and hence a subclass of *transmittable*. Since attribute values in CLOS can only be changed with attribute "accessors," the *defddl* macro sets a trap on all attribute accessors (Lines 4-7) which checks for lock access by calling function *confirm-lock* on the object *before* the attribute value is changed. Hence, if the a transaction tries to update an object that is locked by another transaction, read or write, the access is denied.

Locking enforcement for complex objects is a bit more complex since an object may be implicitly locked because its ancestor is locked (as discussed in Section 3.5). What must be done, therefore, is to modify *confirm-lock* so that it checks for lock access on the object and, if necessary, it goes up the object hierarchy either until it finds an ancestor object that is locked or until it reaches the root of the hierarchy. Recall from Section 3.5 that an object can be updated if and only if one of two conditions is satisfied: either the updating transaction has a write lock (explicit or implied from a locked ancestor) on the object or no transaction has a lock or warning on the object and no transaction has a lock on any of its ancestors. Hence, *confirm-lock* must first check to see if the object itself is locked or has a warning. If it is locked by the updating transaction, then access is granted. If the object is locked by another transaction or if there is a warning on the object, then access must be denied. If there is neither lock nor warning on the object, *confirm-lock* must search for clues the

object's ancestors. If it finds that the calling transaction has a lock on any of its ancestors, then it can proceed to update the object. If, instead, it finds that another transaction has a lock on the object, or one of its ancestors, then it must give up. If *confirm-lock* fails to find any lock on the object's ancestors, then it is assumed to be updating in a cooperative mode (without locks). Therefore, access to the object can be granted.

4.4 DDL Conflict Resolution

Two types of conflicts can be detected by DDL: update conflicts, and lock conflicts. The former are conflicts discovered with the mismatch of DDL update stamps. The latter are conflicts detected when one client session requests for an incompatible lock or tries to update an object what is already locked by another client session. Means must be provided to help users to resolve such conflicts whenever possible. A key difference between these two classes of conflict is that when update conflicts occur, the values of the object replicas may vary from site to site. This is not true with lock conflicts. If the lock conflict has been caused by sequential update messages received out of sequence, by lost lock requests, or concurrent update on the same object version, then the conflict would have been detected by the mismatch of object stamps and indeed treated as an update conflict. Users must intervene to resolve the conflict. This is reasonable since in order to resolve such conflicts, knowledge of the semantics of the operation is needed. Conflicts that are caused by one user trying to obtain an incompatible lock on an object already locked by another user or by one user trying to update an object which is locked for read/write by others, do not cause a mismatch in update stamps since the checking for object access is done at the local site, and is not propagated to other sites until access privilege is granted. With this in mind, we will now discuss the conflict resolution support provided in coSARA.

When an update conflict is detected, all users holding a replica of the object are notified about the object in conflict, the operation that caused the conflict, the user that requested the operation to be performed, and the site that detected the conflict. At this point, the coSARA object-world would collect as much information as possible on the state of the object replica at each site. It would group these sites by the state of their object replica. Sites with matching replica states are in the same group. There can be as many groups as there are sites holding a replica. Replicas are also compared against each other to determine the set of attributes whose values are in conflict.

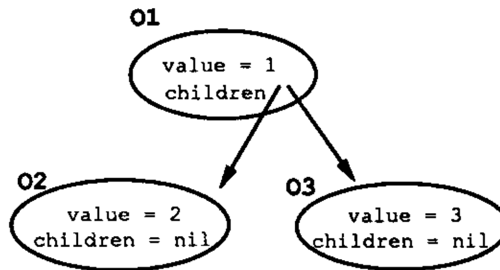
This information is presented to all users holding a replica of the object. Only those users are expected to participate in the conflict resolution process. With this information, users are expected to negotiate, through face-to-face communication or multimedia conferencing facilities, on a "correct" version of the object. When the decision is made, the coSARA object-world will be notified about the location of this "correct" version and will update all replicas of the object with the attribute values found in this "correct" version. If the users

involved cannot agree on an existing version, one user can manipulate an existing replica to get the desired attribute values. Then the coSARA object-world will update all other replicas with the attribute values found in this hand-coded version. When all replicas are finally updated with the correct attribute values, we would again have a consistent state. A scenario of the conflict resolution process is presented in Appendix A.1.

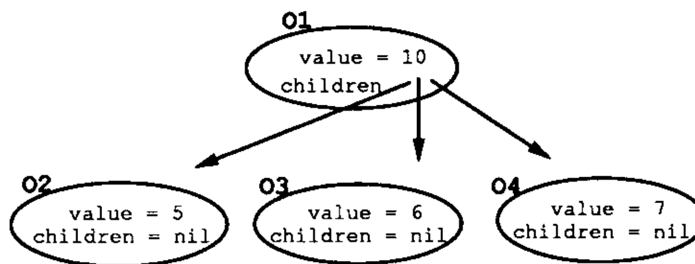
At the database level, "copy-in-place" is used to update the "incorrect" replicas. This is to ensure that all pointers previously pointing to the "incorrect" replica of the object would be kept. This is achieved by requesting the site with the "correct" replica to send to all other sites the ASCII representation of the object. When this is received, each site extracts the attribute values from the ASCII representation (including the DDL update stamp and the lock entry) and updates the local replica of the object accordingly. When this is done, all replicas will have the same attribute values and the same DDL update stamp, yielding consistency. Recall (from Section 4.1) that when an attribute contains pointers to other DDL objects, its ASCII representation is a call to function *read-object* with the storable-id of the referenced object as its argument. What *read-object* does is to find the object in the database with the given storable-id. Since storable-id is a unique and immutable property of the object, it guarantees to return the correct object. The example in Figure 4.7 illustrates this technique. Initially, sites A and B both have a replica of object O1 which agree on the value 1 and "children" elements O2 and O3, as shown in Figure 4.7a. Site A updates object O1 with a new value of 10 and adds a newly created object O4 to O1's list of "children", as shown in Figure 4.7b. It also assigns new values to O1's "children" elements O2 and O3. Because of these changes, objects O1, O2, and O3 all have a new DDL update stamp. However, these update messages were not received by B. After the update conflict is detected, A and B agree to use A's version of O1 as the correct version, and B requests the ASCII representation of A's version be sent to it:

```
"(TREE \"o1\"
       \"RA.asa-105570\"
       (DDL-STAMP . \"S3\")
       (VALUE . 10 )
       (CHILDREN . (list (read-object :id \"RA.asa-105571\")
                         (read-object :id \"RA.asa-105572\")
                         (read-object :id \"RA.asa-105573\"))))
```

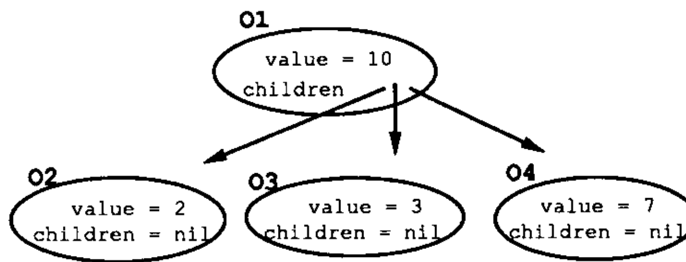
For simplicity, we have assumed objects O1, O2, O3, and O4 to be of class *tree*, a variation of class *btree* as defined in Section 4.1. Further, they have storable-ids "RA.asa-105570," "RA.asa-105571," "RA.asa-105572," and "RA.asa-105573," respectively. When this is received, Site B changes the attribute values of its replica of O1 accordingly, setting O1's "ddl-stamp" attribute to "S3" and O1's value to 10. For the "children" attribute, it calls function *read-object* with the given storable-ids. Since objects "RA.asa-105571" (O2) and "RA.asa-105572" (O3) exist locally, the local replicas of these objects are used. Since object



(a) Sites A and B before the update conflict

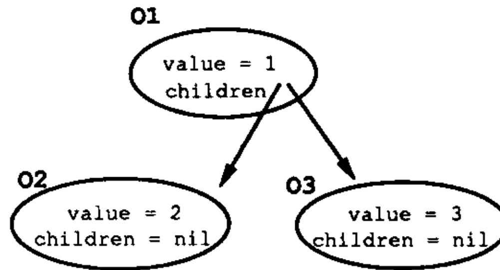


(b) Site A before conflict resolution

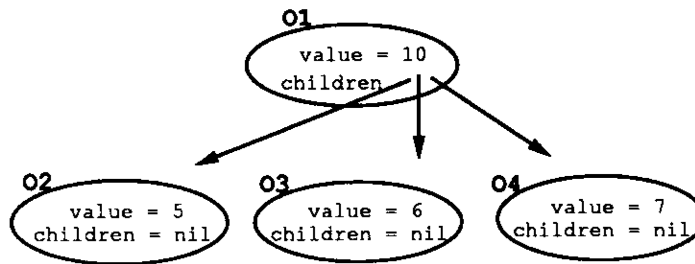


(c) Site B after conflict resolution

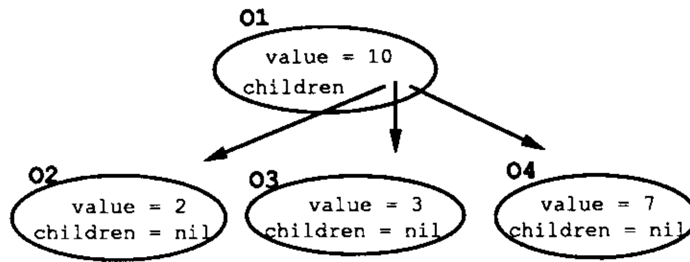
Figure 4.7: Using Copy-In-Place to Resolve Update Conflicts



(a) Sites A and B before the update conflict



(b) Site A before conflict resolution



(c) Site B after conflict resolution

Figure 4.7: Using Copy-In-Place to Resolve Update Conflicts

"RA.asa-105573" was newly created at Site A and does not exist at site B, its values are copied over from Site A. The resulting O1 at Site B is shown in Figure 4.7c. Note that there are differences between A and B's replicas of O2 and O3. However, this is not a problem since these differences will be resolved later, independently, when these two objects are read or updated.

The scheme used to resolve a lock conflict is much simpler. When the lock conflict is detected, the user who initiated the conflicting operation (whether it is a incompatible lock request or a request to update an object locked for read/write by another client session) is notified about the types of locks currently held on that object, and the lock owners. With this information, the user can either choose to give up the operation or to re-try. In the former case, the program is interrupted at the site which requested the lock. In the latter case, the user who initiated the conflicting operation and those who currently hold the locks can negotiate, through either face-to-face communication or multimedia conferencing facilities, for lock access. If the current lock holders agree to release their locks, then the user is given a second chance to perform the operation. Appendix A.2 presents a scenario of lock conflict resolution.

4.5 Network Support for Updating Remote Replicas

The current scheme of forcing all participating sites to listen to and to transmit on the same broadcast channel would not work when the collaborative session spans several networks. This is because, as a general rule, the Internet protocol restricts broadcasting to the smallest possible set of machines [COME88]. Since all updates are broadcast as datagrams in coSARA, this restriction means that no client sites on one network can hear any update messages from another network. Means must be provided to send these update messages to all sites, regardless of their physical locations.

In an effort to implement the most efficient data sharing possible, we have investigated several schemes of packet transmission. A major requirement of the protocol is that it efficiently transport messages between any two client sites on two different networks without degrading the performance of the update broadcast in the purely face-to-face scenario. The simplest but also the most inefficient implementation is to simply send a separate update message directly to each site, as shown in Figure 4.8. This protocol is inefficient at best in the purely face-to-face scenario. Even when all collaborating sites are located on the same network, a separate update message must be sent to each site. Because messages are sent to each site individually, race conditions are likely to occur when more than one site wishes to transmit messages on the network. This removes the guarantee that all sites will receive messages in the same order and therefore increases the likelihood of database inconsistency due to messages being processed in varying sequences at different sites. The advantage of this scheme is that since all update messages are sent individually, it avoids the problem of

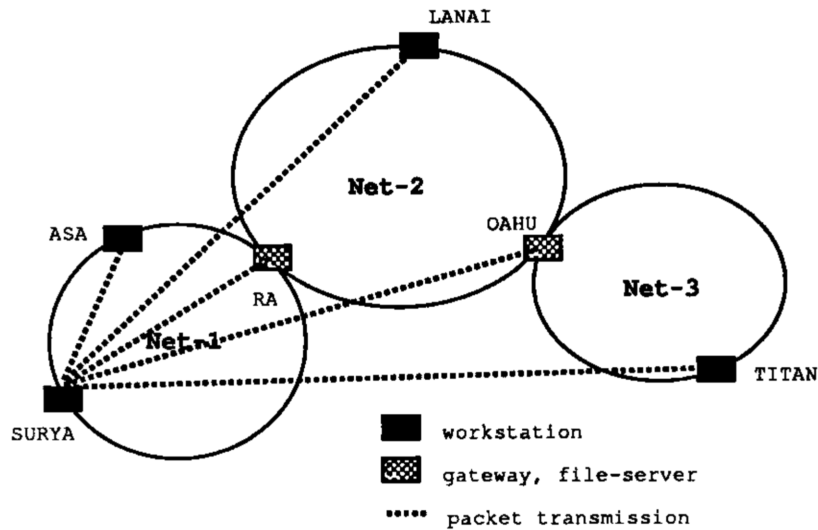


Figure 4.8: Updating Remote Replicas with No Broadcast

a single point of failure and is more robust.

A more efficient method is to have a designated network broadcast server (NBS) on each network. Although each site is responsible for sending each update message to all NBSs on all networks, there is only one NBS at each network that is responsible for relaying the messages it receives from other networks to sites on its own network via broadcast. Whenever a new site joins the collaborative session, it first checks to see if there is another coSARA client site on the network on which it resides. If not, it elects itself to be the designated NBS for that network, and notifies all other sites of this information. Henceforth, all update messages will be broadcast on the local area network where the source site resides, and individually sent to each NBS (except the NBS of the network where the source site resides, of course). Upon receiving such update messages from other networks, each NBS broadcasts the update message on its own network. In this manner, the update message is effectively broadcast on all networks on which a collaborative site resides. An example will clarify this point. We have three interconnected networks – Net-1, Net-2, and Net-3. Site RA logically resides on Net-1 and is in fact the file-server for Net-1. RA also serves as a network gateway between Net-1 and Net-2. As a gateway, it listens to both networks to which it is attached. As a file-server, it can only broadcast messages on the network on which it logically resides. Similarly, OAHU logically resides on Net-3 and is the file-server for Net-3. It serves as a network gateway between Net-2 and Net-3. Suppose we initially have site SURYA in the collaborative session. Since it is the only site running on Net-1, it designates itself as the NBS for Net-1. Site RA joins the collaborative session. It knows that it resides on the Net-1 and that SURYA is the designated NBS for that network. Any update message originating from SURYA is broadcast on Net-1. And since there are no remote NBSs, no other update messages are sent. Similarly, RA broadcasts its updates on the Net-1. Again, since there are no remote NBSs, no other update messages are sent from RA. Since both sites actively listen

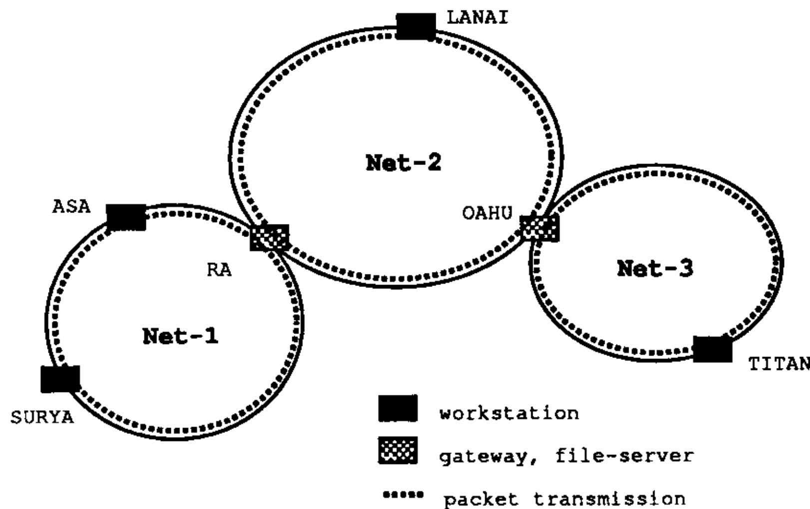


Figure 4.9: Updating Remote Replicas with Broadcast

to the broadcast channel on Net-1, their databases will both be brought up to date with each update message. Now, OAHU from Net-3 joins the collaborative session. Since it is the first site on Net-3 to join the session, it designates itself to be the NBS for Net-3. Both SURYA and RA are notified of OAHU's presents and its status as the NBS for Net-3. And from the object-world server, OAHU knows that SURYA is the NBS for Net-1. When SURYA updates an object, it now broadcasts the update message on Net-1 and sends the same message to Net-3's NBS, in this case, OAHU. RA hears the update message from the broadcast channel. When OAHU receives the message from SURYA, it broadcasts the update message on Net-3 and updates accordingly. If there are no network failures, all three sites would now have an up-to-date copy of the object. On the other hand, when OAHU updates the object, it first broadcasts the update on its own network, and then proceeds to send all NBSs (excluding the NBS for the network on which it logically resides) a copy of the update. Since SURYA is the only other NBS, OAHU sends a copy of the update to SURYA. Upon receipt of this update, SURYA relays this update to all other sites on the Net-1 by broadcasting it. Once again, all three sites would then have an up-to-date copy of the object.

This protocol as described above seems to be relatively efficient. The number of messages sent for each update equals the number of networks with coSARA collaborative sites. However, it has several problems. Consider the following scenario. LANAI on Net-2 now joins the collaborative session. And because it is the first site on Net-2 to join the collaborative session, it elects itself to be the NBS for Net-2. TITAN on Net-3 joins the session as well. Since Net-2 already has an NBS, TITAN does not need to become an NBS. When TITAN updates an object, it broadcasts the update on Net-3 and sends a copy of the update to each of the NBSs on other participating networks, in this case, SURYA and LANAI. Upon receipt of the update message, both SURYA and LANAI broadcast the update on Net-1 and Net-2, respectively. Now comes the problem: Since RA is a gateway and hears messages on both

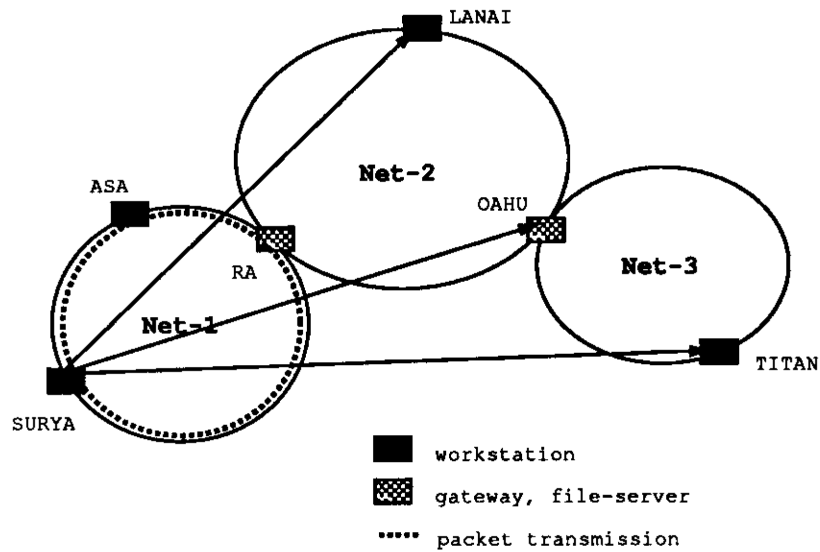


Figure 4.10: Updating Remote Replicas with Broadcast on LAN Only

Net-1 and Net-2, it will receive the update twice, once on Net-1, once on Net-2. Similarly, OAHU will receive the update twice, from Net-2 and Net-3. In other words, means must be taken to filter duplicate messages. One way to do this is to require each client site to announce a network address when it joins the collaborative session and require that client site to listen to broadcast messages on that network only. This is also the network address this site must use to determine if it needs to become an NBS. In our example, if we had required OAHU and RA to listen exclusively to Net-1 and Net-3, respectively, this problem would not have occurred.

There are two major disadvantages to this scheme. First, the NBS may become the bottleneck for update broadcast from remote networks. This is especially undesirable when the participating users are actively updating the database. This would cause messages to be lost and processed out of sequence. Second, the NBS is the single point of failure for the network. If the NBS fails, all client sites on that network would also fail – undesirable indeed.

The model we implemented in coSARA for remote updating is a compromise of the two schemes discussed above. When a site joins the collaborative session, it announces its network address. With this address, and the network address of all other sites in the collaborative session, the new site determines which sites are "remote" in the sense that they are not on the same network as itself. When it updates an object, it broadcasts the update on the local network and sends a copy of the update message directly to each of the "remote" sites. Figure 4.10 clarifies this scheme. We have six sites. SURYA, ASA, and RA announce Net-1 as their network; LANAI announces Net-2 as its network, and TITAN and OAHU uses Net-3. Whereas SURYA considers LANAI, OAHU, and TITAN as its remote

peers, OAHU considers LANAI, RA, SURYA, and ASA as its remote peers. When OAHU updates an object, it broadcasts the update message on Net-3, and sends a copy directly to each of its remote peers. Hence, TITAN receives the update from OAHU's broadcast on Net-3; LANAI, SURYA, ASA, and RA receive the update from OAHU directly. Similarly, when SURYA updates an object, it broadcasts the update message on Net-1 and sends a direct copy to LANAI, OAHU, and TITAN.

There are several advantages to this scheme. First, the performance of this scheme in a purely face-to-face collaborative session is no worse than the performance of a pure broadcast scheme. Second, there is no single point of failure or bottleneck for packet transmission. Third, the sites which serve both as file-servers and gateways will not receive duplicate copies of the update. Its disadvantage is of course its inefficiency as compared to the pure-broadcast scheme.

Although the examples here contain only local area networks on the UCLA Computer Science Department Network, this model would also work with networks outside UCLA Computer Science Department (e.g., DARPA, Perceptronics).

CHAPTER 5

Conclusion

In this paper, a model DDL is proposed to detect all inconsistencies caused by lost messages, messages received out of sequence, and multiple independent updates on a common data version. Because it treats lock requests and releases as updates on the data item, it is also able to detect lock conflicts. A key feature of this protocol is its ability to provide a transparent interface across all modes of data sharing – on the same network or on interconnected networks; cooperatively without locking or coordinated with locks. The only noticeable difference lies in the means of conflict resolution. When users are collaborating in a face-to-face setting, coordination and negotiation for data access can be easily achieved through social interaction. When users are located remotely from each other, coordination and negotiation can be done with multimedia conferencing facilities, FAX, or electronic mail. To make the negotiation process a little smoother, tools are provided in this environment to aid conflict resolution, whether it is conflict caused by lost messages, messages received out of sequence, multiple independent updates on a common data version, or lock conflicts.

In the following sections, we present some of the remaining issues in providing consistency, concurrency control, and remote-sharing to a replicated cooperative environment.

5.1 Extensions to DDL

DDL detects and reports all update conflicts. However, there are times when reporting an update conflict may be unnecessary, particularly if intermediate updates are not needed by the receiving site or if lost update messages are those generated for a locked object by the lock holder. In both cases, the update conflict can be easily resolved by replacing the corrupted replica with a replica from a site which is known to be uncorrupted. Since DDL relies on user intervention for conflict resolution, it is desirable to remove that burden from users whenever possible and to modify DDL to detect and resolve such update conflicts automatically.

Because coSARA supports sharing at the "WYSIWIS" level, conventional protocols which avoid cascading rollbacks fail to work in the coSARA object-world. Because updates do not need to be issued by transactions, the coSARA object-world cannot employ existing

protocols such as strict two-phase locking to guarantee failure recoverability. Although we can minimize data loss by taking frequent snapshots of the database, this is neither efficient nor sufficient. Means should be provided to *guarantee* recoverability in such a replicated cooperative environment.

5.2 Extensions to Communication Support

The current coSARA implementation allows only one client session per machine. This is because each client now maintains its own set of open sockets (say, S0, S1, S2, with port numbers P0, P1, P2, respectively). All clients in the collaborative session communicate on the same port numbers (P0, P1, P2), regardless of their physical location. To have multiple clients on one machine means each of the clients on the machine must maintain a set of open sockets using the *same* port numbers. This is not allowed since each machine port can only be opened once at any given time. Hence, in a configuration with one machine and many X-terminals, only one client can exist, since X-terminals must rely on the machine for its database and communication services. One possible solution is to have one communication server (CS) per machine, such that only the machine CS maintains the set of open sockets on the designated ports. All incoming messages would first be received by the machine's CS, before being transmitted to individual client sessions. All outgoing messages would be addressed to the CS which in turn would put the messages on the designated ports. Hence, the clients would have no knowledge of the ports through which they are communicating; only the CS that is processing messages would have such knowledge.

In our current implementation of the coSARA object-world, we assume all collaborating sites to be on Internet, running TCP/IP, using the same hardware configuration, and running the UCLA CDE software. This assumption is too restrictive. It is highly likely that collaborating users are working on a heterogeneous hardware base, spanning interconnected heterogeneous LANs. With a heterogeneous hardware base, we must make sure that faster machines do not drown slower machines in update messages. With a heterogeneous network, we must consider the impact of varying transport speed on coordination and negotiation.

5.3 Extensions to coSARA Object-World Database Management

In our current implementation of the coSARA object-world, data objects are stored in UNIX file format on our file-server RA. This means that all collaborative sessions must communicate to a centralized database server located on RA (or a machine on RA's local area network). This can be expensive in its performance when users are working remotely from a different network. It would be better if users could have "local" persistent databases on the machines where they work, and at the end of a session, be able to merge the individual

persistent databases. Of course, there are other issues that would have to be dealt with in merging the individual databases.

APPENDIX A

Scenarios of Conflict Resolution

A.1 Scenario of Update Conflict Resolution

Setting: The coSARA group has a computerized group calendar shared by all group members to schedule group meetings. This calendar program is run continuously on the machine display of all group members. Meetings are scheduled no more than 5 working days in advance.

The calendar program operates on five shared objects of class "workday." Each "workday" object consists of 10 attributes, one for each hour from 8:00am to 6:00pm, as shown in Figure A.1. Associated with each attribute is a flag for objection. The flag is initialized to F (for false). When a user issues an objection to a scheduled appointment, this flag is set to T (for true). The five "workday" objects correspond to the five workdays of the week, from Monday to Friday. Figure A.1b is the window used by the calendar program. Each user has this calendar window displayed on his/her screen. It has five buttons, one for each "workday." When a user clicks on one of the calendar buttons with a mouse button, the corresponding "workday" object is displayed on the screen. The calendar program allows three operations:

schedule-appointment: workday, time, business.
cancel-appointment: workday, time.
object-appointment: workday, time.

All three operations change the value of the "workday" argument. Since "workday" objects are shared, changes on one replica will be reflected on all replicas. Operation *schedule-appointment* records the appointment business in the appropriate attribute of the given "workday" object. Operation *cancel-appointment* erases the appropriate attribute value of the given "workday" object. Finally, *object-appointment* sets the "objection" flag from F to T. For simplicity, we assume all appointments to be accepted unless a participant objects to it. Further, we assume that at most one appointment can be scheduled for each hour.

Characters: There are four members in the current collaborative session:

Object **workday**

time	business	obj
8:00am		
9:00am		
10:00am		
11:00am		
12:00pm		
1:00pm		
2:00pm		
3:00pm		
4:00pm		
5:00pm		

(a)

calendar
Monday
Tuesday
Wednesday
Thursday
Friday

(b)

Figure A.1: The "workday" and "calendar" Objects

Dr. Estrin is in his office at 3732E BH, logged in from machine Lanai.
Ivan is working in 3770BH, logged in from machine Asa.
Steven is working in 3770BH, logged in from machine Sol.
Yadran is working in 3770BH, logged in from machine Surya.

Act 1: Dr. Estrin is trying to schedule a group meeting for Tuesday at 10:00am. He makes an entry in the calendar program:

schedule-appointment Tuesday 10:00am group-meeting.

This change is reflected on everyone's calendar program.

Steven, having made prior engagements for that time, objects to the meeting time:

object-appointment Tuesday 10:00am.

This change is also reflected on everyone's calendar program.

Seeing an objection to the scheduled group meeting, Dr. Estrin cancels the appointment for Tuesday:

cancel-appointment Tuesday, 10:00am.

Everyone's calendar is updated, except for Ivan whose machine is having a problem with the Ethernet connection.

Dr. Estrin schedules the meeting for Tuesday at 4:00pm:

schedule-appointment Tuesday 4:00pm group-meeting.

Everyone receives this update, including Ivan. But since Ivan's calendar program failed to receive the cancellation notice for the Tuesday 10:00am meeting, an update conflict is detected. The state of Ivan's "Tuesday" object at this time is shown in Figure A.2b. The

Tuesday

8:00am		F
9:00am		F
10:00am		F
11:00am		F
12:00pm		F
1:00pm		F
2:00pm		F
3:00pm		F
4:00pm	group-meeting	F
5:00pm		F

(a)

Tuesday

8:00am		F
9:00am		F
10:00am	group-meeting	T
11:00am		F
12:00pm		F
1:00pm		F
2:00pm		F
3:00pm		F
4:00pm		F
5:00pm		F

(b)

Figure A.2: State of "Tuesday" at Time of Update Conflict

```
Update Conflict Detected for Tuesday
Operation: schedule-appointment
Issued by: estrin@lanai
Detected by: ivan@asa
Inconsistent Attributes:  10:00am, 4:00pm
Site Groups: ("Lanai" "Surya" "Sol") ("Asa")
```

Figure A.3: Window Notifying Users About the Update Conflict

state of everyone else's "Tuesday" object at this time is shown in Figure A.2a. Therefore, everyone is notified about the update conflict, with a window popping up on their screen, as shown in Figure A.3.

Dr. Estrin gets on the speaker phone with the group members in 3770BH.

Dr. Estrin : Ivan, what do you have scheduled for Tuesday at 10:00am?

Ivan : The group meeting.

Dr. Estrin : Hmm. That has been changed. What do you have scheduled for Tuesday at 4:00pm?

Ivan : Nothing according to the calendar.

Dr. Estrin : Is Tuesday at 4:00pm OK for everyone?

Yadran : Actually, no. I promised to take Sebastian to the Chocolate Factory at that time. How about Tuesday at 2:00pm?

```
Access Denied for Wednesday
Operation: schedule-appointment
Issued by: yadran@surya
Lock Holder: ivan@asa for WRITE
```

Figure A.4: Window Notifying Users About the Lock Conflict

Everyone present agrees that is a good time for the group meeting.

Dr. Estrin : Let's do it.

Steven : I'll change the calendar.

Steven hand-codes his copy of the Tuesday calendar, removing the entry for Tuesday 4:00pm and adding an entry for Tuesday at 2:00pm.

Steven : Ready.

Ivan : OK, here it goes.

Ivan enters the site name "Sol" at the prompt. Now everyone sees the meeting scheduled for Tuesday at 2:00pm.

A.2 Scenario of Lock Conflict Resolution

Act 2: Ivan has the group calendar's Wednesday entry locked for write. Yadran needs to schedule his dissertation defense dry-run on that Wednesday:

schedule-appointment: Wednesday, 10:00am.

But Yadran receives an error message, as shown in Figure A.4:

Yadran : Ivan, what are you doing with the Wednesday calendar?

Ivan : Oh, I had it locked to record my next vacation. Do you need it?

Yadran : What do you think is more important, your vacation or my defense? Ha. Justs kidding. Please release the lock when you are done.

Ivan releases the lock.

Ivan : OK. I have released the lock.

Yadran trys *schedule-appointment* again. This time, access is granted. Two weeks later, Mr. Yadran became Dr. Yadran.

Bibliography

[ABDE88] Abdel-Wahab, H.M., Guan, S., Nievergelt, J. Shared Workspaces for Group Collaboration: An Experiment Using Internet and UNIX Interprocess Communications, *IEEE Communications Magazine*, November 1988, pp.10-6

[BANN90] Bannon, L.J., Schmidt, K. CSCW, or What's in a Name? by personal communiation, June 1990

[BERN81] Bernstein, P.A., Goodman, N. Concurrency Control in Distributed Database Systems, *Computing Surveys*, Volume 30, No. 2, June 1981

[BERN84] Bernstein, P.A., Goodman, N. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases, *ACM Transaction on Database Systems*, Volume 9, No. 4, December 1984, pp.596-615

[BERN87] Bernstein, P.A., Hadzilacos, V., Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987

[BERT87] Bertsekas, D., Gallager, R. *Data Networks*, Prentice Hall, 1987

[CASN90] Casner, S., Seo K., Edmond, W., Topolcic, C. N-Way Conferencing with Packet Video, *Proceedings of the Third International Workshop on Packet Video*, March 22-3, 1990

[CERI84] Ceri, S., Pelagatti, G. *Distributed Databases: Principles And Systems*, McGraw Hill, 1984

[COME88] Comer, D., *Internetworking with TCP/IP*, Prentice Hall, 1988

[CROW90] Crowley, T., Milazzo, P., Baker, E., Forsdick, H., Tomlinson, R. MMConf: An Infrastructure for Building Shared Multimedia Applications, *Proceedings of the Conference on Computer-Supported Cooperative Work*, October, 1990

[DAVI89] Davidson, S.B. Replicated Data and Partition Failures, *Distributed Systems*, ed. Mullender, ACM, pp.265-92, 1989

[DIED89] Diederich, J., Milton, J. Objects, Messages, and Rules in Database Design, *Object-Oriented Concepts, Databases, and Applications*, ed. Kim and Lochovsky, ACM Press, 1989, pp.177-98

[DITT86] Dittrich, K.R. Object-Oriented Database System: The Notions and Issues, *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, ed. Dittrich and Dayal, IEEE Computer Society Press, 1986

[ELLI90] Ellis, C.A. A Model and Algorithm for Concurrent Access Within Groupware, by personal communication, October 1990

[ELLI91] Ellis, C.A., Gibbs, S.J., Rein, G.L. Groupware: Some Issues and Experiences, *Communications of the ACM*, January 1991, pp. 39-58

[ESTR86] Estrin, G., et al. SARA (System ARchitect's Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. *IEEE Transaction on Software Engineering*, February, 1986

[FORS85] Forsdick, H. Exploration into Real-time Multimedia Conferencing *Proceedings of the Second International Symposium on Computer Message Systems*, September 1985, pp.331-347

[FOST87] Foster, G. *Collaborative Systems and Multi-user Interfaces: Computer-Based Tools for Cooperative Work*, PhD dissertation, Computer Science Division, University of California atBerkeley, Report No. UCB/CSD 87/326

[GREE89] Greenberg, S., Chang, E. Computer Suport for Real Time Collaborative Work, *Proceedings of the Conference on Numberical Mathematics and Computing*, Winnipeg, Manitoba, September 1989,

[GREE91] Greenberg, S. Computer-Supported Cooperative Work and Groupware: an Introduction to the Special Issues, *International Journal on Man-Machine Studies*, 1991, pp.133-141

[GREI86] Greif, I., Seliger, R., Weihl W. Atomic Data Abstractions in Distributed Collaborative Editing System, *Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, pp.160-172, 1986

[GREI87] Greif, I., Sarin, S. Data Sharing in Group Work, *ACM Transactions on Office Information System*, April 1987, pp. 187-211

[JAJO89] Jajodia, S., Mutchler, D. A Pessimistic Consistency Control Algorithm for Replicated Files which Achieves High Availability, *IEEE Transaction on Software Engineering*, Volume 15, No.1, January 1989, pp. 39-46

[KEEN89] Keene, S.E. *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989

[KNIS90] Knister, M.J., Prakash, A. DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors, *Proceedings of the Conference on CSCW*, October 1990, pp. 343-55

[KORT86] Korth, H.F., Silberschatz, A. *Database System Concepts*, McGraw Hill, 1986

[KUNG81] Kung, H.T., Robinson, J. On Optimistic Methods for Concurrency Control, *ACM Transaction on Database Systems*, Volume 6, No. 2 June 1981 pp.213-226

[LANT86] Lantz, K.A. An Experiment in Intergrated Multimedia Conferencing, *Proceedings of CSCW '86 Conference on CSCW*, Austin, 1986, pp.267-275

[MUJI91] Mujica, S.T., *A Computer-based Environment for Collaborative Design*, PhD dissertation, UCLA Computer Science Department, 1991

[NIER89] Nierstrasz, O. A Survey of Object-Oriented Concepts, *Object-Oriented Concepts, Databases, and Applications*, ed. Kim and Lochovsky, ACM Press, 1989, pp.3-22

[PARK83] Parker, D.S., et al. Detection of Mutual Inconsistency in Distributed Systems, *IEEE Transactions on Software Engineering*, May 1983, pp.240-247

[PATT90] Patterson, J.F., et al. Rendezvous: An Architecutre for Synchronous Multi-user applications, *CSCW '90 Proceedings*, October 1990, pp. 317-28

[POPE90] Popek, G.J., et al. Replication if Ficus Distributed File Systems, *IEEE Technical Committee on Opreating Systems*, Volume 4, No. 3., 1990

[SARI88] Sarin, S., Greif, I. Computer-based Real-Time Conferencing Systems, *Computer-Supported Cooperative Work: A Book of Readings*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988, pp.397-420

[SCHO89] Schooler, E.M., Casner, S.L. A Packet-switched Multimedia Conferencing System, *ACM SIGOIS Bulletin*, January 1989, pp.12-22

[SILB80] Silverschatz, A., Kedem, Z. Consistency in Hierarchical Database Systems *Journal of the ACM* Volume 27, No. 1, January 1980, pp.72-80

[SKAR89] Skarra, A.H., Zdonik, S.B. Concurrency Control and Object-Oriented Databases, *Object-Oriented Concepts, Databases, and Applications*, ed. Kim and Lockovsky, ACM, pp. 395-422, 1989

[SON89] Son, S.H., Argrawala, A.K. Distributed Checkpointing for Globally Consistent States of Databases, *IEEE Transaction on Software Engineering*, Volume 15, No. 10, October 1989, pp. 1157-1167

[STEE90] Steele, G.L., Jr. *Common Lisp: The Language*, 2nd Edition, Digital Press, 1990

[STEF87] Stefik, M., Foster, G., Bobrow, D.G., Lanning, S., Suchman, L. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meeting, *Communications of the ACM* Volume 30, No. 1, pp.32-47, 1987

[TANE89] Tanenbaum, A.S. *Computer Networks*, Prentice Hall, 1989

[TOML89] Tomlinson, C., Scheevel, M., Kim, W. Sharing and Organization Protocols in Object-Oriented Systems, *Journal of Object-Oriented Programming*, November/December 1989, pp.25-36

[TRAI82] Traiger, I.L., Gray, J., Galtieri, C.A., Lindsay, B.G. Transactions and Consistency in Distributed Database Systems, *ACM Tansactions on Datbase Systems*, Volume 7, No. 3, September 1982, pp.323-342

[ULLM88] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, Volume I, Computer Science Press, Maryland, 1988

[WALP91] Walpole, J., Yap, M. Concurrency Control, Version Management and Transactions in Advanced Database Systems, Oregon Graduate Institute Technical Report No. 91-009

[WOLF87] Wolfson, O. The Overhead of Locking (and Commit) Protocols in Distributed Databases, *ACM Transaction on Database Systems*, Volume 12, No. 3, September 1987, pp.453-471