AN ABSTRACT OF THE THESIS OF

Denise J. Ecklund for the degree of Doctor of Philosophy in
Computer Science presented on December 16, 1986.

Title: Robustness in a Distributed Storage Server for Engineering Design Data with Versions

Redacted for Privacy

Abstract approved: _____
Fred M. Tonge

Traditional database management systems have been deemed unsuitable for use in the Computer Aided Engineering environment. Object-oriented data models with project management features have been proposed as an alternative. The Distributed Hypothetical Storage System is intended to serve as the underlying storage mechanism for an object-oriented data model. The storage system provides features that are essential to project management such as version histories, alternate data versions, and optimistic concurrency control in a distributed environment.

This thesis presents a robustness mechanism for DHSS. Three classes of system failure are considered: site failure, communication failure, and media failure. The protocols described here make the system robust in two senses: first, users are provided with continued read access to their data; and second, users are provided with continued write access to their data. The traditional approach of data replication is used to support continued read access. However, traditional systems support no write access or highly restricted write access in the event of a failure. The DHSS robustness mechanism supports continued write access for all visible data by allowing the execution of possibly conflicting write requests and attempting to merge or resolve the results at a later time if they are indeed conflicting. The resolution protocol requires one exchange of information among the communicating sites followed by unilateral calculation to resolve all conflicts existing among that set of sites. It is essential that the resolution be mutually consistent among the set of communicating sites. It is shown that the robustness protocols produce a mutually consistent resolution of conflicting requests.

Robustness in a Distributed Storage Server for Engineering Design Data with Versions

by

Denise J. Ecklund

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed December 16, 1986

Commencement June 1987

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Chairman of department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented December 16, 1986

Typed by researcher for Denise J. Ecklund

ACKNOWLEDGEMENTS

I dedicate this work to my mother Shirley Hollenbeck, and in loving memory of my father Roy Hollenbeck. Their many years of love and encouragement are of unmeasurable value to me.

I extend my sincerest thanks to Fred Tonge, my advisor. I am most grateful for the constructive comments he gave me, for his willingness to listen to an idea, and for his continued encouragement. He possesses that rare quality of leadership combined with an attitude conducive to a team effort. I am pleased that we will have the opportunity to be colleagues. I look forward to our future joint work.

From September 1984 through December 1986 I held a Fellowship position in the Computer Research Lab of Tektronix Laboratories. I would like to thank Kevin Considine and Rick LeFaivre for awarding me this position thus permitting me the opportunity to work and grow in the fine research environment they have established.

Thanks to Rick Krull and Bob Eifrig who worked on the implementation of the DHSS prototype. They helped to point out the implementational considerations of my protocols. Thanks to Jim Blick, my consulting statistician, for all those helpful hours on the telephone. I wish to thank many people at the Computer Research Laboratory for their help and encouragement: Larry Morandi for the insightful troff hacks, Jim Alexander for his consultation on the experimental designs, and the many others from CRL who showered me with encouragement and reference materials.

Last, but not least, I thank my wonderful husband Earl. Without his continual encouragement I would have given up many years ago. With his support I have completed this work and have achieved a confidence to build on for the rest of my life. Thank you Earl for believing in my abilities.

TABLE OF CONTENTS

LIST OF FIGURES

ROBUSTNESS IN A DISTRIBUTED STORAGE SERVER FOR ENGINEERING DESIGN DATA WITH VERSIONS

INTRODUCTION

This thesis presents a robustness mechanism as part of a distributed storage system for storing versioned data. The proposed use of the storage system is as a component of a database system for computer-aided engineering (CAE). In this introduction we discuss the evolution of database applications, the requirements for a CAE database system, and why current database systems are inadequate for use in the CAE environment. We propose a layered architecture for a CAE database system which includes as a component a robust distributed storage system. We conclude this introduction with an overview of the structure of the thesis.

1.1 Database applications

The study and design of database systems began early in the 1960's with the work of Charles Bachman [8]. By the 1970's, the applicability and design of distributed databases were being investigated. The database research of almost two decades centered on various forms of business data processing as the targeted application area. In the late 1970's, many new application areas for database technology were perceived. This gave rise to database research in medical information systems, decision support systems, management information systems, statistical database systems, and computer-aided instruction. It quickly became clear that the results produced from 1962 - 1978 were not sufficient for solving the problem of database support for these new application areas. In particular, the hierarchical data model [56,77], the Codasyl (DBTG) network data model [41,73], and the relational

data model [18,22] are not semantically powerful enough to model these new problems. Much effort has gone into extending these models [23] to capture facets of the new application areas such as statistical databases [62], temporal databases [21], and Management Information Systems [71]. The one principal common to all these database systems was the use of a hierarchically layered architecture. The conceptual layer encapsulated the user's view of the data; the logical layer was the database system designer's view of the data; and the physical layer described how the data and the relationships among the data are represented on the physical storage devices.

1.2 Databases for Computer-Aided Engineering

In the early 1980's a number of researchers proposed using database support for engineering design [47,64]. Existing computer-aided engineering systems consist of a disparate collection of tool programs. A tool program may assist the engineer in editing a schematic of a design, testing a design by simulating the functionality of the design, or testing the consistency of several component designs by testing the consistency of their respective interfaces. The major problem with the current set of tools is that each uses a different data representation. Thus, the engineer must also be equipped with a set of conversion programs to convert the output of one tool into the required input format for another tool. Significant time is spent performing data format conversion. The problem is compounded by the fact that the engineer must have explicit knowledge of the data format requirements of each tool and must ensure that those requirements are meet. In contrast, a single data base system could be used to store all design data and to encapsulate all representation conversions required by the various tools. Such a database system would also provide reliable storage of design data, controlled access to the design data, the maintenance of complex relationships among the components of a design, automatic consistency testing of a design, storage of alternate designs, a consistent interface for users, minimal service interruptions, and easy addition of new tools to the system.

We believe that the users of a CAE database system would interact with the system in the following manner. If an engineer wishes to work on a design he may use an edit tool. The tool program would ask the database system for access to a copy of the design the user

wishes to work on. The database system would provide the tool program with a private copy of the requested data in proper external format. The user makes modifications to the design using the tool program. When all modifications have been completed the edit tool requests that the database system store the revised design in the database. The database system is responsible for testing the consistency of the design and reliably storing the new design in the database. From the user's view this sequence of steps would be considered a transaction [78] (i.e. the single execution of a single program).

1.2.1 Attributes of CAE as an Application

From this interpretation of how users would interact with a CAE database system we have extracted the following attributes of CAE as an application area.

1. A single user level transaction may extend over hours or days [47].
2. Design data has an arbitrary internal format.
3. The quantity of data affected by an update is large [47].
4. The relationships among pieces of design data are very complex [6].
5. The set of users would be divided into many relatively small design teams.
6. Many of the users would access the system via workstations in their own office [6, 48, 81].
7. Design data must be available regardless of other users activities [64] or system failures.
8. Users must be able to investigate and store alternative designs.
9. Users must be able to revert to a previous version of a design.
10. Because the users work as a team only minimal authorization must be enforced by the system [64].

The attributes of CAE are seen in contrast to those of a data processing application where user level transactions are short-lived, small amounts of data are updated per transaction, only one version of each data item exists at any given time, data has a highly structured format, the system has a large number of users, and the intentions of many of the users are questionable. Clearly CAE as an application area differs significantly from those application areas investigated previously.

1.2.2 A Proposed Architecture for a CAE Database System

Several research groups are designing and implementing portions of a CAE database system [2, 6, 45, 48, 81]. Figure I.1 shows a proposed five layer architecture for a CAE database system. Each layer provides services to the layers above it in the hierarchy. The hierarchical structure attempts to reduce the complexity of designing and building the database system itself.

The user application layer is a collection of CAE tools such as a graphic editor, a design rule checker, and a simulator. The representation interface layer [2] is concerned with the mapping between internal and external format of the design data stored by the system. Configuration management [14] builds a single complex entity called a configuration by selecting and combining versions of subcomponents of the entity. At the data model layer, use of an object-oriented data model [2, 38, 58] to enforce consistency of the design is proposed. The storage server provides reliable storage of the design data. Below the storage server is the local operating system.

The users interact with the layered system by executing a tool program. The tool program would ask the database system for access to a copy of the design the user wishes to work on. When a request for data is received, the configuration layer will determine exactly which pieces of data are being requested. The storage system will locate and provide the requester with a private copy of each data item. The representation layer will convert the data to the format appropriate to the requesting tool. The user then works with his copy of the data using the operations provided by the tool program. When all user modifications have been completed the tool will request that the database system store the revised design in the database. The representation layer will format the external representation of the

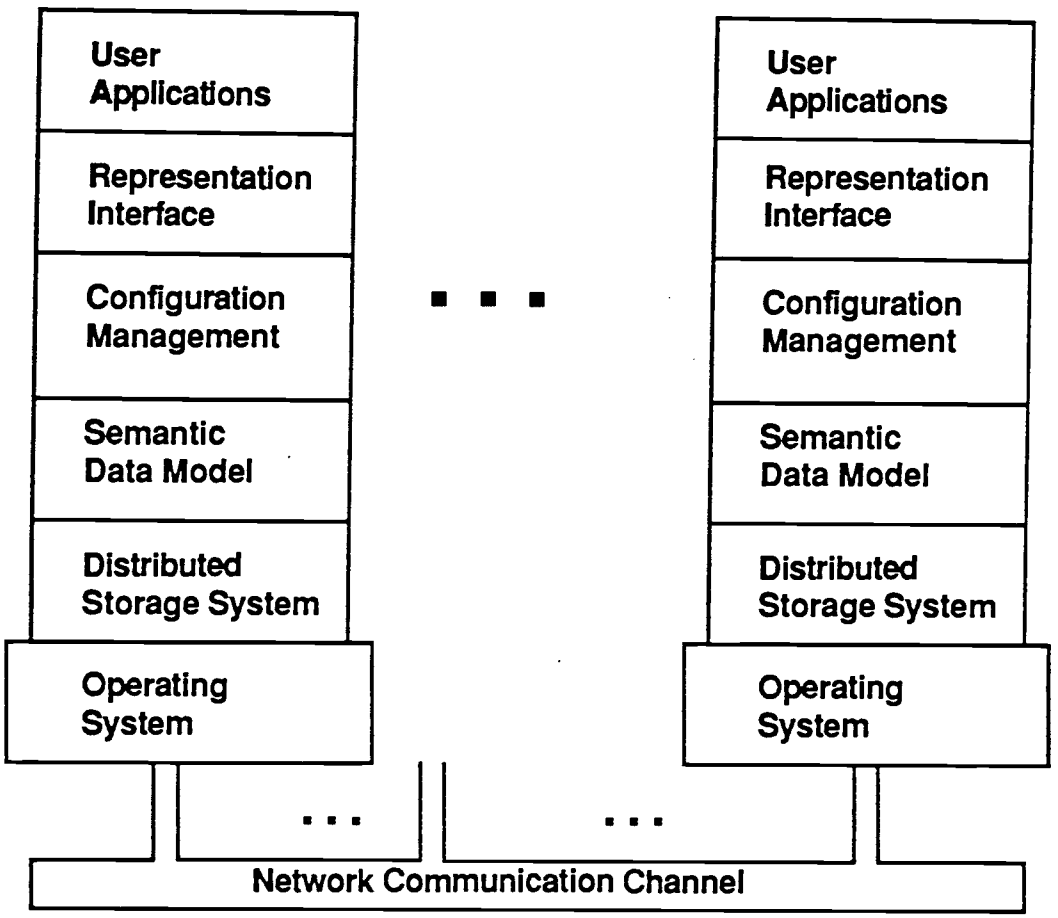


Figure I.1 - Five Layer Architecture for a CAE Database System.

revised design into its corresponding internal format. The configuration layer will determine which subcomponents have been modified and if necessary, modify the configuration definition. The data model layer will certify the modified design by invoking appropriate consistency checking programs. Once the modified design has been certified it will be stored in the database.

1.3 Requirements for the Distributed Storage System

Given the five layer architecture and the system wide attributes the following requirements have been established for the storage system.

1. All data stored by the storage system will be typeless.
2. A storage model appropriate for project-oriented applications will be supported by the storage system.
3. Distributed multi-reader and multi-writer access will be supported.
4. Internal consistency and mutual consistency of system information will be maintained.
5. During failure of any system component, maximum read and write access for user data will be supported.

1.4 Overview of this thesis

This thesis presents the design of a robust distributed storage system appropriate for use as the fifth layer of the CAE System architecture. A robust system is one that executes transactions even when components of the system have failed. The set of all failures can be divided into four classes: site crashes, network partitions, media failures, and software failures. The robustness mechanisms proposed for the storage system will manage site crashes, network partitions, and media failures. Software failures will be discussed in an overview on the four failure classes but no solution to the problem of software failure will be presented. The robustness protocols presented in this thesis have been implemented as part of the Distributed Hypothetical Storage Server.

This thesis is organized in nine chapters. Chapter two defines criteria for a robust distributed database system. We present the four failure modes for a distributed database system and the problems posed by these failures. We examine what has and has not been done in managing these failure modes.

Chapter three presents a distributed project-oriented storage model [31,32]. Versioning is the basic relationship modeled by the storage model. We propose this model for the storage system layer of the CAE database architecture. The storage model is implemented by a distributed storage server. A subset of the operations supported by the storage server are described. The basic processing structure of the server and its mechanisms to support multi-reader/multi-writer user access are discussed. Three other version servers are presented. We compare and contrast these storage models with our system.

In chapter four we define the phrase "optimistic robust distributed system". We define an optimistic access rule for the storage system and show how that rule can be supported by employing techniques such as replication of user data, virtual partitions, pseudo name server sites, and pseudo primary sites. We discuss each of these techniques in detail and show how they can be integrated with our distributed storage system.

Chapter five presents our approach to making the distributed storage system robust to three of the four failure classes defined in chapter two. The technique of virtual partitions is used to detect site or communication failures and to detect the correction of site or communication failures. When failures are detected, message propagation is used to maintain mutual consistency among a set of communicating sites. When failure correction is detected, a merge protocol is performed to regain mutual consistency among the set of communicating sites.

Chapter six presents a detailed discussion of the merge protocol required to regain mutual consistency among communicating sites in our distributed storage system. By supporting the optimistic access rule, conflicting requests may have been processed by sites that were unable to communicate. The merge protocol must propagate the results of non-conflicting requests, and resolve the results of conflicting requests. Conflicts are divided into two classes: direct conflicts and merge-able conflicts. Pairs of DHSS requests are classified as direct conflicts or merge-able conflicts. Merge algorithms are presented for resolving each pair of conflicting DHSS requests. It is shown that the merge recovery algorithms maintain mutual consistency among a set of communicating sites.

Chapter seven presents two examples of the merge resolution algorithm applied to conflicting group updates. Each example consists of two consecutive merge recoveries, illustrating how one merge recovery may effect subsequent merge recoveries.

Chapter eight presents the design of an experiment to evaluate the usefulness of multi-reader/multi-writer systems versus single-reader/single-writer systems. The usefulness of each system is measured by the lost time users incur while using each of the systems. We hypothesize that a multi-reader/multi-writer system incurs less lost time than a single-reader/single-writer system; therefore, a multi-reader/multi-writer system is more useful. The proposed experiment could be performed by a direct experiment using the DHSS prototype or by building a simulation.

Chapter nine summarizes what we have learned from designing and prototyping a robust multi-reader/multi-writer storage system. We propose possible future work that would enhance the distributed storage system and augment the robustness protocols.

CHAPTER TWO
ROBUST DISTRIBUTED DATABASE SYSTEMS

The purpose of a database system is to reliably store data and to process valid and authorized read and write requests on that data. An additional benefit to the user is that the database system provides a uniform interface for accessing data by executing transactions. A transaction is a unit of work which consists of a sequence of steps required to achieve some goal. Each transaction consists of an initiation phase; a read, compute, and write phase; and a commit phase. In the initiation phase the transaction is authenticated and assigned some priority for being executed. In the read, compute, and write phase the transaction carries out the requested action. The commit phase installs all of the results written by the transaction into the database making them visible to other transactions.

The purpose of a distributed database system is to provide global database service to users at a collection of sites. A distributed database system is a coalition of sites that share a common view of and control access to a set of data. We require that each site maintain directory information about the data so that each site has knowledge of all data stored by the database system. Copies of the data itself are distributed among the sites. There may exist multiple copies of any data item. The sites are connected by a communication network consisting of some physical communication medium and a communication protocol for controlling communication over that medium. This basic communication protocol is part of the network services of the operating system.

A distributed database must, under normal operating conditions, maintain the internal consistency and the mutual consistency of the data stored by the system. Internal consistency is maintained if all the updates performed by committed transactions are reflected in the database, none of the updates performed by the uncommitted transactions are reflected in the database, and no transaction is allowed to read a data value written by another transaction that is not yet committed [39]. A distributed database system maintains mutual consistency of the database if given a data item, all replicated copies of that data item are the same. More specifically Thomas defines mutual consistency to mean, "all copies converge to the same state and would be identical should update activity cease" [75].

2.1 Robustness

A robust distributed database system, as defined by Eager and Sevcik [28], is one that processes transactions and preserves internal and mutual consistency even when system components have failed. We define four types of component failure: site crash, network partition, media failure, and software failure.

1. Site crash: Site crash is a machine failure due to loss of power, operating system deadlock or panic, processor malfunction, or human intervention to restart a site. In general a site crash is indistinguishable from a network partition.

2. Network partition: A network partition occurs when two or more groups of sites are running but are unable to communicate. A partition may occur due to a site crash or a failure of the communication channel. A failure of the communication channel is defined to be any failure beyond the recovery control of the basic communication protocol provided as part of the network service of a typical operating system.

3. Media failure: Media failure occurs when a storage device fails while reading or writing data rendering some portion of the stored data unreadable.

4. Software failure: Software failure occurs when the internal consistency or the mutual consistency of the database has been compromised due to an error in the implementation of the database system or due to the occurrence of failure types not managed by the protocol implemented by the software.

Preserving internal consistency in the event of failures requires some mechanism for undoing results written by uncommitted transactions and ensuring the results written by committed transactions are reflected in the database. Preserving mutual consistency in the event of failures in the distributed environment requires that all sites acquire knowledge of new data items and new values for existing data items in as timely a manner as possible. Besides preserving internal and mutual consistency during failures we contend that a robust database system must provide database users with a graceful degradation in database access and a reasonable recovery of services when failures are corrected. Database access in a distributed database degrades gracefully if the percentage of transactions that cannot be processed due to component failures increases proportionately with the decrease in the

number of functioning system components. We say that the system provides the user with reasonable recovery from site crash if the user is notified of the outcome of any interrupted transactions. This can be accomplished by a reply mechanism external to the storage system that notifies the user of results at a later time.

Many researchers have investigated the four failure modes (site crash, network partition, media failure, and software failure) and their effects on internal and mutual consistency and graceful degradation and recovery. Typically failure modes have been investigated in isolation; in particular, consistency has been investigated without regard to degradation of access or the gracefulness of recovery. A more detailed discussion of the failure modes and some interesting results on protocols for dealing with them are presented below.

2.2 Implications of Site Crash and Network Partitions

Failures usually result in the isolation or the loss of data stored by the system thus failures inherently limit access to data. A distributed system supporting graceful degradation of access will attempt to minimize the limitation on access due to failures.

2.2.1 Read Access and the Effects of Data Replication

Limitations on read access are minimized by replicating the data stored by the system. If one copy is inaccessible due to failure, a second or third copy may be accessible. This requires strategic placement of copies. Evaluating a particular placement algorithm is very difficult. One must evaluate the cost of initially placing the data, the cost of one site retrieving a data copy from another site, and the cost of updating multiple copies on multiple sites. A model for the pattern of retrieval requests and system failures must be developed to evaluate the benefits of replication. A number of algorithms for the automatic placement of data replicas have been proposed and analyzed [15, 16, 20, 55, 61]. They provide solutions for

different network and application environments; as each approach assumes different cost/benefit factors.

Most placement algorithms assume a long haul network[1] where physical distance between sites, the number of sites that must store and forward the message as it moves from its source to its destination, and the dynamically changing routes that may be taken from the source to the destination are all key factors in the cost of reading data. In contrast, the distributed operating system LOCUS[2] proposes a solution to the problem of replicating data in a local area network. LOCUS allows the user to decide dynamically the number of copies to be made and where those copies should reside [80]. Under the assumption that the user knows how, when, and where the data will be used this approach could yield better retrieval performance than automated placement.

2.2.2 Write Access and the Effects of Concurrency Control

Limitations on write access are controlled by the application's definition of conflicting transactions. Every transaction has an associated read set containing the data items read by the transaction and a write set containing the data items written by the transaction. It is assumed that the read set contains the write set. Two transactions are said to conflict if the intersection of the write set of one transaction with the read set or write set of the other transaction is non-empty. For performance gains a database system will interleave the execution of transactions. If two transactions conflict the execution of one must precede the execution of the other in order to maintain internal consistency. For two conflicting transactions T_1 and T_2, T_1 precedes ($\rightarrow$) T_2 if any operation of T_1 conflicts with an operation of T_2 and is executed before any operations of T_2. An interleaved execution is serializable if the results produced by the execution are the same as the results produced by some sequential execution of the transactions [10]. Serializability is a sufficient condition for an acyclic $\rightarrow$ relationship among all conflicting transactions. The purpose of a concurrency control protocol is to ensure the serializability of the system's interleaved execution. An

[1] A long haul network is one in which some sites are separated by more than 10 km and the sites function as store and forward nodes for sending messages [60].

[2] LOCUS is a Unix compatible distributed operating system developed at the University of California at Los Angeles.

interleaved execution is correct if it is serializable and eventually all transactions are either processed or rejected.

Many concurrency control mechanisms have been proposed. These mechanisms may be divided into three categories: protocols based on locking [1, 68, 72], protocols based on timestamps [11, 65, 75], and optimistic protocols [17, 50].

1. Locking Protocols: Lock based systems require that transactions request locks on data items at read time. Once locks are obtained they are held until the transaction completes its writes, then they are released. (This is known as two-phase locking [35].) Transactions may deadlock waiting to obtain locks on an overlapping set of data items. If a lock is currently held by a transaction T_i and transaction T_j requests the lock, transaction T_j enters a transaction_wait_for relationship with transaction T_i. Transaction T_i in turn may be in a transaction_wait_for relationship with another transaction. If a cycle exists in the transaction_wait_for relationship a deadlock has occurred. Thus, deadlock avoidance or deadlock detection with transaction backout must be part of a locking protocol.

2. Timestamp Protocols: In a timestamp based system, every transaction is assigned a unique timestamp and every data item has an associated write timestamp. Transactions are allowed to perform reads and writes on data items if the timestamp associated with the data item is less than the timestamp of the transaction. When a data item is updated its associated timestamp is set equal to the timestamp of the writing transaction. If a transaction cannot write its results because its timestamp is too small, the transaction must be restarted with a new larger timestamp.

3. Optimistic Protocols: Optimistic protocols allow transactions to execute freely reading data and writing copies of the data. A consistency check for serializability is performed before committing a transaction's write set to the database. If the consistency check fails the transaction is backed up and restarted. The hope is that few transactions will conflict thus back up and restart will rarely be required.

2.2.3 Robust Optimistic Concurrency Control

Basic concurrency control assumes a failure free environment. The processing of writes is restricted during site or network failures because the acyclicity of the $\rightarrow$ relationship may be violated if writes were processed by non-communicating sites. For a CAE database system such restrictions on writes would result in the loss of many hours of productivity for a number of users. This is unacceptable. Two different approaches have been taken that allow all write requests to be processed during site and network failures.

1. Every transaction is assigned a unique number establishing an ordering on the transactions. If a site has executed some transactions out of order then undo those transactions and redo them in order using standard concurrency control for controlling the execution sequence.

2. Redefine the concept of conflicting transactions so that fewer transactions conflict and resolve the conflicts that do occur by a merge based on the semantics of the update operations.

Allowing updates to be processed during failure conditions is most easily viewed as an extension to optimistic concurrency control. The hope is that few conflicting updates will occur if failure conditions do not persist for extensive periods of time.

A robust optimistic concurrency control mechanism is an optimistic concurrency control mechanism that allows all reads and writes to be processed regardless of which sites are or are not able to communicate. Several researchers have investigated such protocols. Popek, Thiel, and Kline proposed a mechanism for conflict detection and semantic merge. Davidson proposed a mechanism for conflict detection, backout to a non-conflict state, and re-execution of the conflicting transactions. Sarin, et. al. proposed an optimistic protocol that uses optimism as the general mode of operation rather than a strategy for processing during failures. These three approaches to optimistic concurrency control are discussed in detail below.

2.2.3.1 Version Vectors in LOCUS

Popek, Thiel, and Kline [63] proposed the use of version vectors for maintaining mutual consistency of replicated files in the LOCUS system [79]. A version vector is associated with each copy of a file. A version vector is a list of pairs, one pair for each site that holds a copy of the file. These pairs contain the name of a copy site and a count of the number of file updates that were initiated from that site. When a file F is created at a site S_1, k copies of F are placed on sites S_1, S_2, ..., S_k and the version vector $<S_1:1, S_2:0, ..., S_k:0>$ is associated with F on every site S_1, S_2, ..., S_k. If S_2 initiates an update of file F, then every site S_i that acquires a copy of the latest version of F from S_2 will increment the update count associated with S_2 producing the version vector $<S_1:1, S_2:1, S_3:0, ..., S_k:0>$. If site failures or communication failures occur updates are allowed to proceed. When the site or communication channel is back in service a loss of mutual consistency can be detected by comparing the version vectors associated with each copy of a file. A set of version vectors are said to be compatible if "one vector is at least as large as any other vector in every site component for which they each have entries" [63]. This largest version vector is associated with the version of the file that is the most current and should be propagated to all other copies of the file. If a set of version vectors are not compatible then mutual consistency has been lost due to conflicting updates.

LOCUS performs some automatic merging of conflicting updates for files of special types such as file system directories and mailboxes. This is possible because the only operations defined on directories and mailboxes are insert and remove. File system directories are merged by considering each directory entry as an extension of the file it corresponds to. When a directory entry is inserted or removed, the version vector of the associated file is incremented. When a merge occurs, if one version of a file is determined to be the current version of the file then that file version and its associated directory entry will be propagated to other sites. For update conflicts among untyped data objects LOCUS retains all versions and reports the inconsistency to a higher level user for merging.

Two advantages of the version vector approach are that the algorithm for maintaining version vectors is simple and the extra storage required for a vector is small in comparison to the size of a typical file it is associated with. One shortcoming of the version vector proposal is that when a read request is made you must query all the sites known to hold a

copy of the file in order to locate the most recent version. The alternative to this is to extend the protocol to keep track of the site failures and network partitions and automatically compare version vectors and perform propagation when the failures are corrected. It would also be desirable to extend the protocol to manage semantic merging of file system directory entries updated by operations such as "change owner" or "change access permission". Such an extension would require that a version vector be associated with each individual directory entry. This would greatly increase the extra space for vectors required in implementing this approach.

2.2.3.2 Global Precedence Graphs

Assuming an environment where all writes are allowed in all partitions, Davidson proposed the use of a precedence graph for detecting sets of conflicting updates and the removal of conflicts by performing backout and redo on a set of transactions [25]. The precedence graph for the merger of partitions P_1 and P_2 contains a node for each transaction executed in partition P_1 and a node for each transaction executed in partition P_2. The nodes are connected by three types of edges: ripple edges, precedence edges, and interference edges. A ripple edge from node t_i to node t_j means that transaction t_j read a data value D written by transaction t_i, there is no other transaction t_k such that t_k wrote D after t_i wrote D and before t_j read D, and t_i and t_j were executed in the same partition. A precedence edge from node t_i to node t_j means that transaction t_i read a data value D that was later written by transaction t_j, t_i and t_j were executed in the same partition, there is no ripple edge from node t_i to node t_j, and there is no transaction t_k such that t_k wrote D after t_i read D and before t_j wrote D. An interference edge from node t_i to node t_j means that t_i and t_j were executed in different partitions, t_i read a data value D that was written by t_j in the other partition. Conflicts are represented by cycles in the precedence graph. The conflicts are broken by selecting transactions for backout and removing the corresponding node from the precedence graph. If node t_i is selected for backout all nodes t_j connected to t_i by a ripple edge must be selected for backout. Selecting the minimum number of transactions for backout is an NP-Complete problem. Davidson proposes an enumeration of the cycles in the precedence graph. As each cycle is discovered transactions are selected for backout until the cycle is

broken. This approach is polynomial in the number of cycles in the graph which may be exponential in the number of nodes.

There are several shortcomings to this approach. Large amounts of storage are required to store the entire processing history of each partition. When two partitions merge the size of the processing history grows to be the sum of the merging history sizes. Furthermore subsequent merges are not independent of previous merges; the transactions selected for backout in one merge may be present in the processing history of a partition in a future merge and such transactions must be removed from the merging partition's history. One final problem is that the protocol does not include a method for keeping track of the partition and merge behavior of the set of sites.

2.2.3.3 Autonomous Concurrency Control by Timestamps

Sarin, Blaustein, and Kaufman have proposed an optimistic concurrency control protocol that allows writes to be executed with no distributed verification of serializability [69]. Their system consists of three modules running on every site, the interface, the distributor, and the checker. The interface accepts updates from users and presents the users with data. The distributor provides a reliable broadcast protocol ensuring that all sites will eventually receive notification of all updates. A variant of Awerbuch and Even's algorithm [7] for reliable broadcast in an eventually connected network is used. The distributor receives locally and remotely generated updates and logs them for processing by the checker. Each update request has a unique associated timestamp which creates a total ordering on all updates. The checker reads new update requests from the log, inserts them into the general processing log in timestamp order, and perform a series of log transformations to determine how to integrate all new updates with those already processed by the site. The valid log transformations are based on the time commutativity of non-conflicting transactions and the arithmetic commutativity of some types of updates. After all log transformations have been performed the checker can determine all sets of conflicting transactions. Those conflicting updates which have been previously executed must be rolled-back so that the set of conflicting updates can be processed in timestamp order. All updates which are not members of a conflict set are processed. This protocol does not

perform atomic transactions in the traditional sense; because no form of locking is used, users may read values in intermediate state produced by concurrently executing update operations. When updates are re-processed side effects of any previous executions might have to be undone. An alerter tests for side effects due to previous executions. If the alerter detects a side effect that must be undone a trigger program is executed to take compensating action.

The positive aspects of this protocol are that updates can arrive at any time and be processed in the global timestamp order, there is no central control site , and detection of site crash or network partition is not required for the proper functioning of the system. Disadvantages to this protocol are that a continuous log of information on the processing of every request must be kept indefinitely. The authors do outline a plan to discard the oldest log information at most of the sites keeping a complete log at only a few designated backbone sites. This policy would defeat the intent of the protocol as it introduces the possibility of not being able to process an update due to an inability to obtain log information from a backbone site. A second possible problem with the protocol is that compensating actions performed by trigger programs can lead to oscillating behavior. If, due to a network partition, a group of sites G is delayed in receiving an update U and G has executed a conflicting update with a larger timestamp than U's associated timestamp, then all sites in G will backout U, the alerter may note an associated side effect from a previous execution of U and all sites in G will independently initiate a compensating action. When the update reflecting the compensating action arrives at other sites in G the other sites must recognize that multiple compensations have taken place and may attempt to correct the situation by compensating for the double compensation.

2.2.4 Tracking Site Crashes and Network Partitions

The robust optimistic methods proposed by Popek, et. al. and Davidson require that site crashes and network partitions be recognized. Given that sites may be slow due to local system load and communication channels may be bogged down transferring large amounts of data it is impossible to declare that a site is down or that the network is partitioned with 100 per cent assuredness. Furthermore, before the failure is repaired, it is impossible to distinguish between a site crash and a partition that isolates a single site.

2.2.4.1 Virtual Partitions

El Abbadi, Cristian, and Skeen proposed a protocol that manages views on physical partitions [33]. A view on a physical partition is called a virtual partition. A virtual partition is a list of sites that have agreed that they can talk to each other and will talk only to those sites in the list for the processing of transactions. The list of sites may not be correct in that it may contain sites which are not reachable or it may fail to contain sites which are reachable. The virtual partition protocol allows the site list to be modified by agreement when evidence is given that the list is incorrect. Two events which serve as evidence that the site list is incorrect are: receiving a message from a site not in your local site list or failing to receive acknowledgement of a message sent to a site in your local site list. Any number of mutually exclusive virtual partitions may exist simultaneously.

A two-phase protocol is used to construct the site list for a virtual partition. One site functions as an initiator by sending a message to all known sites inviting the site to join a new virtual partition. Each virtual partition has a unique name and the initiator site proposes a name for the virtual partition. Each site that receives an invitation must either reply with an acceptance message or begin initiating another virtual partition. A site accepts the invitation if the proposed virtual partition name is larger (in some ordering) than the name of the virtual partition the site currently belongs to. Eventually some set of reachable sites will agree to join a virtual partition with the largest proposed name. The initiator sends, to all sites that accepted the invitation, a commit partition message containing the new site list.

Sites functioning within a virtual partition must communicate only with those sites in the partition site list. Each data item written by a site has the local site's current virtual partition name written with it. Tagging data items with the virtual partition name signifies that the value being written was known only by those sites belonging to the virtual partition named. Whether or not a data item can be written depends on whether or not the virtual partition contains the quorum required by that data item [34]. Quorums are used to insure a serializable execution of requests. When two virtual partitions merge to form one virtual partition, every site independently calculates what data items will have a quorum in the newly formed virtual partition. Each site must obtain a copy of the current value for each data item that will be writable in the new virtual partition. This is accomplished by requesting a copy of the data item and its partition tag from each site in the new virtual partition; the value with the largest associated partition tag is saved as the new value.

One drawback to this protocol is that negotiating a new virtual partition can produce a large number of network messages before a partition name is agreed upon. The events that trigger the negotiation of a new virtual partition are inherently asynchronous. Thus, in the worst case, for a set of N sites, $(N^2 - D)$ messages may be sent to negotiate a virtual partition containing (N - D) sites. One potential problem with this protocol is that if failures and the repair of those failures occur cyclically at just the proper time intervals every site may loop executing the negotiation and data value acquisition protocol. The probability of such behavior occurring is unknown.

2.2.5 Site Crash and Internal Consistency

A site crash may occur at any time during the processing of a set of transactions. If any transaction has written some but not all of its results the internal consistency of the database has been lost. Solutions to this problem involve the use of a write ahead log written to stable storage [39,40]. Stable storage is any storage medium that usually survives a site crash.

A log is a stream of records that describe the processing of each transaction. The log contains information on who requested the transaction, what the request was, the before and after values of each data item written during the execution of the transaction, and

information on events that occurred during the processing of the transaction, which may be of interest to recovery. When a site comes up following a crash, a restart phase is initiated to bring the database to a locally consistent state. The information in the log will be used to undo any partially executed transactions and restart them, to verify that the results written by a completed transaction have been successfully written to stable storage, and to notify the user who issued the request of the outcome of the transaction. To support undo/redo activity the log entries pertaining to a single transaction must be physically connected in some way and they must be physically written before the execution of the events they report. The write ahead nature of a log requires support of a forced or synchronous write facility to guarantee that the log is written before the results of a transaction.

In many database systems the log is retained indefinitely. Clearly this requires large amounts of stable storage space. IBM's[3] Information Management System[4] (IMS) used streaming tape as a logging medium [57]. This provided the system with an endless amount of stable storage space but performing synchronous writes to a tape drive was indeed slow. IBM's Database 2 system[5] used magnetic disk as the storage medium to log current transactions while log entries pertaining to older committed transactions are migrated to streaming tape [24]. This approach supplied the endless stable storage of a magnetic tape and the increased access speed of the disk device. Database systems for business applications write logs containing sufficient information to represent a formal audit trail of the system's transaction processing. An audit trail records information on what user requested the transaction, what data the user entered or retrieved, what action was requested, how the request was processed, and the before and after values of data items updated due to the request [13].

In contrast to an endless audit trail, Reuter proposed an undo logging scheme using shadow pages and an over-writable log file [66]. In shadow paging [54] the database is divided into logical units called pages and each logical page is represented by two physical pages on a magnetic disk device. One physical page holds the current working version of the logical page; the second is a shadow page that holds the previous and backup version of the logical page. During execution of a transaction all writes of a logical page are performed on the working version physical page. When the transaction is committed the current

[3] International Business Machines Corporation

[4] Information Management System is a trademark of International Business Machines Corporation.

[5] Database 2 is a trademark of International Business Machines Corporation.

working pages become the shadow pages, freeing the old shadow pages to be used as current working pages by other transactions. Reuter's shadow paging scheme allocates two contiguous physical pages p_i and p_i' per logical page L_i. Each physical page contains space reserved for a timestamp consisting of the system clock time and a transaction id. Each transaction is assigned a unique id thus every timestamp is unique and there exists a total ordering on the timestamps. When a transaction writes a working page the appropriate timestamp is constructed and written with the page. When a transaction reads data contained in L_i the system reads both p_i and p_i' and examines the timestamp on each page to determine the current page. A rewritable undo log keeps track of all uncompleted transactions and their associated transaction id. When a transaction completes it is removed from the undo log by freeing the log space for reuse.

As an undo mechanism this scheme works as follows. If a site crash occurs all physical pages of the database must be scanned. If a page has been written by a transaction identified in the undo log the timestamp associated with that physical page is set to zero. This has the effect of making the shadowed page become the most recent version of the page. After the database has been scanned all undo log entries can be freed. Providing automatic redo of interrupted transactions requires that additional information about the original request be written to the over-writable undo log.

The main advantage of this approach over others is speed of execution. The time spent writing updated pages is the same, the time spent reading is increased only slightly by reading two contiguous pages rather than one page, and the time spent logging is much less because so little information is logged and only two synchronous writes are required per transaction (one prior to initiating execution of the transaction and a second before returning the committed result to the user). The disadvantage of this approach is the amount of stable storage space used. Even though the log space required is small the database storage requirements have doubled. Also all requirements for stable storage must be meet by magnetic disk devices.

The logging techniques discussed above are useful for regaining local internal consistency following a site crash. They are not sufficient for regaining mutual consistency among a set of sites following a crash of one or more of the sites.

2.3 Media Failure

A media failure results in a loss of information. The only way to recover the lost information is to have multiple copies of the information or an encoding of the information from which the lost information can be regenerated. Checkpointing combined with transaction logging have been used as a method for maintaining multiple copies of a database. Both are expensive mechanisms in terms of time and space. A checkpoint is a backup copy of the entire database; it is created when the database is in a consistent state. The transaction log contains information on all updates executed since the checkpoint was created. (A checkpoint and an abbreviated transaction log could be used in place of an endless transaction log for purposes of regaining local internal consistency following a site crash.)

Chan, et. al. [19] propose a scheme where each site creates local checkpoints, maintains local transaction logs, and recovers from its own media failure. If a site incurs a media failure it is assumed that only one of the three entities (the database, the checkpoint, or the transaction log) were destroyed. The occurrence of a media failure while recovering from a media failure is not managed.

Kuss [51] proposed transaction synchronized checkpoints of each site but when a media failure occurs all sites participate in the reconstruction of the failed site's database. In this case multiple media failures may be recoverable but some transactions may be discarded when transaction logs are destroyed.

Attar et. al. [4] proposed constructing a consistent checkpoint of the entire database at one or more sites. The checkpoint is constructed by a transaction which reads and copies the entire database. The consistency of the checkpoint was guaranteed by the concurrency control mechanism.

2.4 Software Failure

Software failure is a very difficult problem and has been studied in one form as the problem of Byzantine agreement [5, 26, 27] in which a set of processors attempt to reach an agreement when some of the processors are faulty. The problem of Byzantine agreement assumes that

1. all sites begin with a value of V,
2. the value of V can be altered only by executing a deterministic agreement protocol,
3. if all non-faulty sites have an original value of V = x then the final decision reached by all non-faulty sites must be that V = x,
4. all communications are performed in broadcast mode,
5. faulty sites may behave in any manner but they do not effect the decision of the non-faulty sites,
6. the decision reached by a faulty site is inconsequential.

A bound on the number of messages required to achieve Byzantine agreement has been obtained under the assumptions that sites are synchronous (i.e. all sites keep their local clock within Φ time units of every other site's clock) and communications are synchronous (i.e. any message sent by the local site will be received within time Δ). These assumptions imply that timeouts are reasonable approximations of site or communication failures. If sites and communication are asynchronous there is no bound on the number of messages required to reach an agreement.

These results are insightful in that they demonstrate the power of synchronous behavior and the impossibility of dealing with asynchronous behavior. However they do not constitute a solution to our problem of software failure. Assuming synchronous behavior (which is a big assumption), we must specify a decision procedure for concluding that a software failure has occurred, determine which site is at fault, and provide a mechanism for correcting any database inconsistency that may have resulted due to the faulty site. It is questionable whether or not a decision procedure exists that will always detect a faulty site.

Consider the case where site S may be faulty and some site D sends a request message to S. If site S does not reply within time ($\Delta + \Delta + \Phi$) or site S gives a detectably invalid reply, site D concludes that site S is faulty. However, it may be the case that S is not faulty and gave a valid reply, but site D is faulty and does not recognize S's reply as valid. Site D then attempts to get all other sites to agree that site S is faulty. There may be no reasonable way for the other sites to test whether or not site S is indeed faulty. Site S may reply in a valid manner to any message posed by another site whether or not S is faulty in some other way. In any event, the basic problem remains that sites S and D were not able to complete some transaction between them and this may have caused a loss of mutual consistency in the database. From a correctness point of view, it is more important to detect the possibility of an inconsistency and correct it than to know which site(s) was faulty and caused the inconsistency. For performance reasons it is desirable to be able to determine which site is faulty so that the situation may be corrected rather than repeatedly attempting to recover from inconsistencies created by a site's faultiness.

2.5 Components of a Robust System

The goals of a robust system are to provide the users with continuous service while maintaining internal consistency and mutual consistency of the database. The traditional approach to building a robust system has been to extend the failure free protocols for maintaining internal consistency and mutual consistency. The more successful extensions for maintaining consistency include write ahead logging, two-phase commits, and timestamps. Write ahead logging is used to regain internal consistency in the event of site crash. Two phase commits ensured the mutual consistency by synchronizing writes at all sites in the system. Timestamps ensured mutual consistency by ordering the writes performed by each site in the system. The extensions for providing continuing service during failures include data replication, quorum-based distributed concurrency control, and conflict detection and merge protocols. Data replication provides continued read service to users in the event of site or communication failure. Quorum-based concurrency control allows those updates for which a quorum exists to be processed even though sites or

communications have failed. Conflict detection and merge protocols allow all updates to proceed employing merge techniques for resolving update conflicts as they are discovered.

From our discussion it is clear that degrees of robustness can be gained only at considerable cost in time and space. It is also the case that we can be robust only in the event of failures we can detect and have anticipated detecting. We propose a combination of new and old robustness mechanisms in an attempt to achieve an acceptable degree of robustness at a smaller cost in time and a significantly smaller cost in space.

CHAPTER THREE
A DISTRIBUTED STORAGE SYSTEM FOR PROJECT-ORIENTED APPLICATIONS

Building a robust system requires a detailed analysis of the class of failures the system will manage, the environment the system will be used in, the basic assumptions made by the users of the system, and the data model, storage model, and processing model used by the system itself. In chapter two we discussed the three failure classes we will address in building a robust storage system: site crash, network partition, and media failure. We now present our assumptions concerning the environment for the distributed storage system and the data, storage, and processing models used by the storage system.

3.1 The DHSS Environment

The environment for the CAE database system consists of a collection of workstations and possibly larger machines connected by one or more local area networks. Based on the need for distributed access to the system it was decided that the storage system should provide all aspects of distribution for the entire system, making distribution as transparent as possible to the other layers of the database system. By distributed system we mean a collection of autonomous machines connected by some communication channel; each machine processes requests and stores data.

Figure III.1 shows a possible CAE database environment with multiple local area networks connected by gateway sites. Each local area network connects a number of machines. Project teams of five to fifteen engineers spread over the network will access the system through a workstation in their office or through a larger machine. The workstations may or may not be single user machines. It is assumed that every machine has a local hard disk of some minimal size. The larger machines will gives users access to the system and will be designated as fileserver sites.

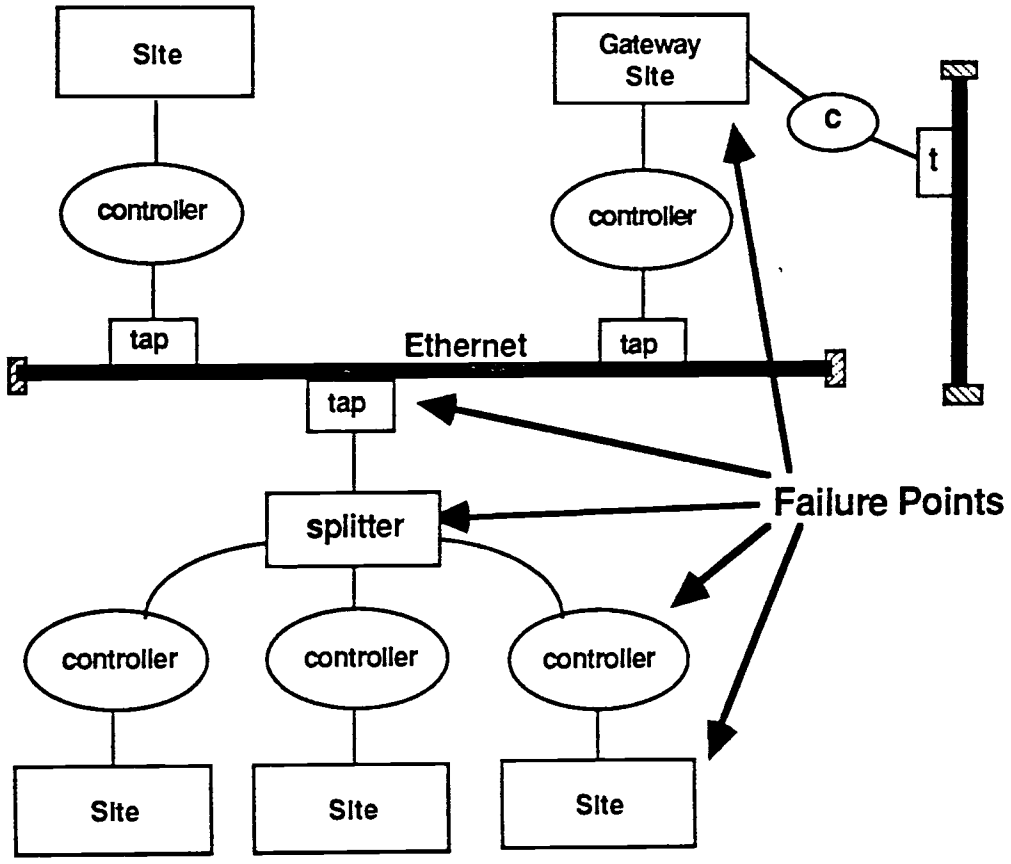


Figure III.1 - A CAE Database Environment and its Failure Points.

The communication medium for each local area network (LAN) is an Ethernet[6]. An Ethernet provides unreliable broadcast communication among the sites connected to it. We assume the existence of an operating system network facility that supports a virtual circuit mechanism over the Ethernet. A virtual circuit gives the user the appearance of error free communication between two sites. Messages are received in the order they are sent and the non-deliverability of a message is reported to the sender. There is a small probability that the network facility may incorrectly report a message as undeliverable.

Each site has a network controller which receives messages that are broadcast over the Ethernet medium. In order to maximize the number of sites connected to a single Ethernet, some controllers are connected to a splitter. The function of the splitter is to pass incoming messages from the Ethernet on to all attached controllers. When a site sends a message, the

[6] Ethernet is a registered trademark of Xerox Corporation.

message is sent by the controller through the splitter and out onto the Ethernet. Every network controller must be connected to a splitter or an Ethernet tap.

Each Ethernet tap, each splitter, each network controller, and each site are possible failure points in the environment. As discussed in chapter two, the failures we consider are fault-stop failures where the malfunctioning device stops service. Some Byzantine failures, where the device continues to function but in a faulty manner, are detected as breaches of protocol, such as a bad checksum or an incorrect reply in a multi-phase protocol. In general, Byzantine failures will not be considered in our environment.

We assume that a site failure is a site crash followed, at some time in the future, by a restart of that site. A failed site may also have experienced a media failure due to the site crash. Permanent removal of the site is also possible.

If a controller fails the site connected to that controller cannot communicate with the remainder of the network. This single site forms one physical partition and the remaining sites form another. The sites that continue to communicate are unable to determine if the failed site has crashed or has been partitioned by a communication failure.

If a splitter or a tap fails each site connected to the Ethernet by that tap or splitter becomes a single site partition and the remaining sites form another physical partition. It is important to note that a splitter generally has no local loop-back capabilities. Two sites attached to the same splitter can communicate with each other only by sending a message out over the entire Ethernet proper. Thus, if a splitter fails each site connected to the splitter forms a single isolated partition.

A site that connects multiple local area networks is called a gateway site. If a gateway site fails, two partitions are formed. We will call such partitions probable partitions. The important characteristic of a probable partition is that the maximum set of sites that will be members of the partition is already known.

3.2 Federations

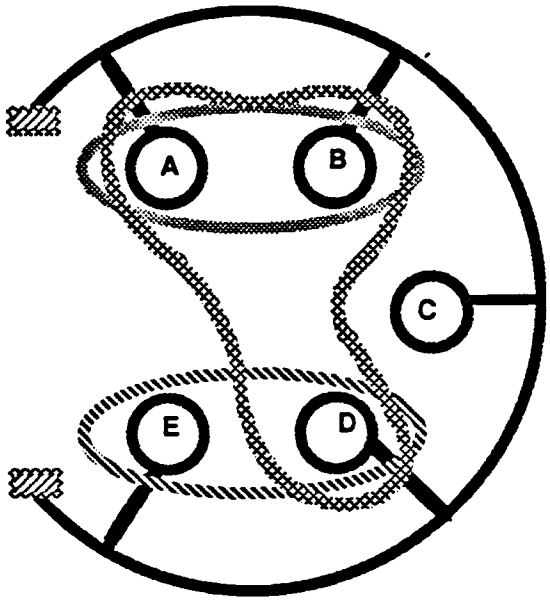
In order for users to interact with each other and to share their data there must be a global mechanism for organizing users and data in a meaningful way. A federation is such a mechanism [31]. A federation is a loosely associated collection of sites, users, and the data they share. Formally, a federation is defined by an ordered triple <O, S, U>, where O is the set of objects in one scheme defined by the data model, S is the set of sites that belong to the federation, and U is the set of users who belong to the federation. For all future discussion the term user and the term client will refer to the upper layers of the database system which act as clients to the storage system; the term end-user will refer to the human engineer who will use the database system.

If one user at a site is a member of a federation that site is also a member of that federation. A user may be a member in any number of federations. Membership in a federation is associated with permission to access the data stored in that federation. There are two modes of membership in a federation. A user may be an associate member or a participating member. A participating user may modify the database by creating or updating entities in the object set. Each entity created is owned by the creating user. Associate users can only read the objects stored by the federation. Sites also hold participant or associate status. If any user at a site is a participating member of a federation then that site is a participating member of that federation.

All sites belonging to a federation will maintain fully redundant directory information about the sites, users, and data objects in that federation. Participating sites will maintain copies of the data extensions for the set of objects. For reliability multiple sites may be asked to store full copies of the data extensions. Machines with larger local disks are the preferred sites for placement of data copies but due to the possibility of site or communication failures such placement may not always be possible. Given that the majority of the machines will be workstations with local disks of minimal capacity a migration or archival technique must be employed to relieve the workstations of a possible storage burden.

Each federation has an associated name that may or may not be unique. A federation is named when it is defined by one user at one site. There is no meta-federation name server to approve federation names, hence users at other sites may define distinct federations

having identical names. Figure III.2 shows sites A, B, C, D, and E connected by a local area network and grouped into the federations named Big, Little, and Little.



Federation Name	Members
Big	A,B,D
Little	A,B
Little	D,E

Figure III.2 - Sites A, B, C, D, and E Participating in Federations Big, Little, and Little.

3.3 Federation Management

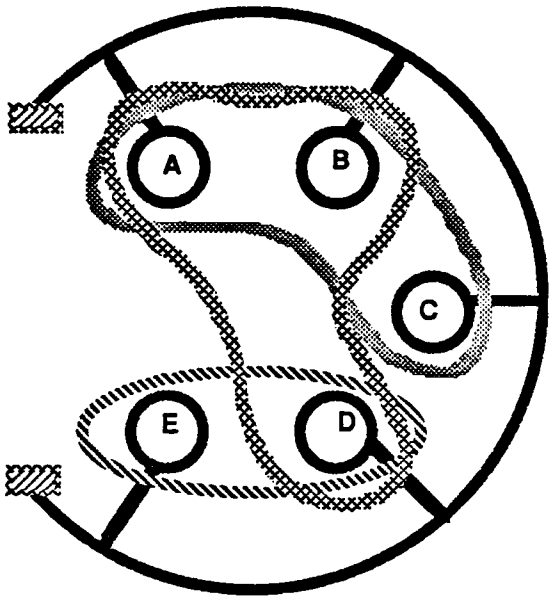
The process of managing a federation is a dynamic one. User initiated actions and environmental system failures may at any moment in time cause changes in the active structure of a federation. The user initiated operations which affect the set of sites and the set of users in a federation are define, enroll, and secede. A user may define a new federation on the local site, enroll in a federation which exists on another site, or secede

from a federation in which the user is currently a member. The execution of each of these operations may be affected by site crashes and network partitions.

Define is a user initiated operation which creates a new federation. The initiating user must give a name to the federation being created. This proposed federation name must be a unique federation name on the local site. If the local site already belongs to a federation of the specified name the new federation cannot be defined and the user is informed. If the federation name is unique then a new scheme <O, S, U> is created. The site set contains the local site, and the user set contains the initiating user. The local site becomes the designated name server site for the new federation. The user and the local site are automatically considered to be participating members in the new federation. The object set is empty. Optionally the defining user may specify a default replication factor for the object set of the new federation.

The enroll operation allows a user to join an existing federation. If the site local to that user does not currently belong to this federation the site is enrolled also. The user initiating the enroll must give the name of the federation which is to be joined, the name of a site known to belong to that federation, and the participation mode the user wishes to enroll under. It is necessary for the upper layers to provide the name of a remote site in an enroll request due to the possibility of duplicate federation names. In this case the distributed nature of the storage system is seen by the upper layers of the system. Figure III.3 shows the federations Big, Little, and Little of figure III.2 with site C enrolling in the federation named Little that site A belongs to. A successful execution of the enroll operation will result in the initiating user being added to that federation's set of users. If the local site was not a member of the federation at the time of the enrollment, then the local site will be added to that federation's set of sites. Since the local site will not have a directory for federations to which it does not belong, a copy of the directory for this federation will be obtained. The mechanics of acquiring the directory during an enroll is handled as a normal part of the robustness mechanism and will be discussed in chapter five.

An enroll operation may fail for many reasons. It is possible that the local site and the specified site already belong to different federations of the same name. In this case it is impossible for the user to enroll in the desired federation. The enroll operation will also fail if the specified site can not be reached due to nonexistence of the specified site, site failure,



Federation Name	Members
Big	A,B,D
Little	A,B,C
Little	D,E

Figure III.3 - Site C Enrolls in Federation Little on Site A.

or network failure. In such a case the user could try specifying another site, if another site is known to belong to the desired federation.

The secede operation is used when a user wishes to withdraw from a federation. A user may secede from a federation only if he is no longer the owner of any entities of the object set. If all of the local users of a federation secede from that federation the site also secedes from that federation. The seceding site is obliged to retain all redundant copies of object data and sufficient directory information to access these copies until they have migrated or have been archived on a remote dedicated device.

3.4 A Distributed Storage Model for the Object Set

E. Ecklund and D. Price [32] proposed a single site storage model for the object set of a federation. The storage model reflects all of the features critical to data management in any team design environment: support for a high degree of data sharing; mutual consistency of sets of data objects; storage and retrieval of temporal versions of data objects; and creation, storage, and retrieval of alternate versions of data objects. Their storage model has been implemented by the Hypothetical Storage Server (HSS) which runs on a 4.2bsd UNIX[7] system. The single site storage model was extended to a distributed storage model [31] and implemented by the Distributed Hypothetical Storage Server (DHSS) [29].

3.4.1 Structure of the Storage Model

DHSS tracks the evolution of a design, storing and maintaining access to temporal and hypothetical versions of each design. The use of hypothetical versions allows a designer to investigate alternate but related designs. Each alternate version is related to either the principal version of a design or an alternate version of the same design. This means that at the configuration management layer each entity maps to a design component and at the storage level that design is represented by DHSS as a tree of instantiations. Such a tree is called a DHSS object. The root of the tree is the first instantiation (first version) of the design to be stored in the system. One path in the tree is denoted as the principal path; the instantiations along this path correspond to each recorded update on the principal design (i.e. the temporal versions of the principal design). Every other path in the tree corresponds to an alternate design; the instantiations along such a path are the temporal versions of the alternate design. For each path in the tree there is a distinguished node which corresponds to the current version (with respect to time) in that path. Figure III.4 show a DHSS object containing four version paths, the principal path and three alternate paths.

The DHSS data base consists of a collection of DHSS objects. In the real world these objects may be subcomponents of each other or may be semantically related in some way.

[7] UNIX is a registered trademark of AT&T Bell Laboratories.

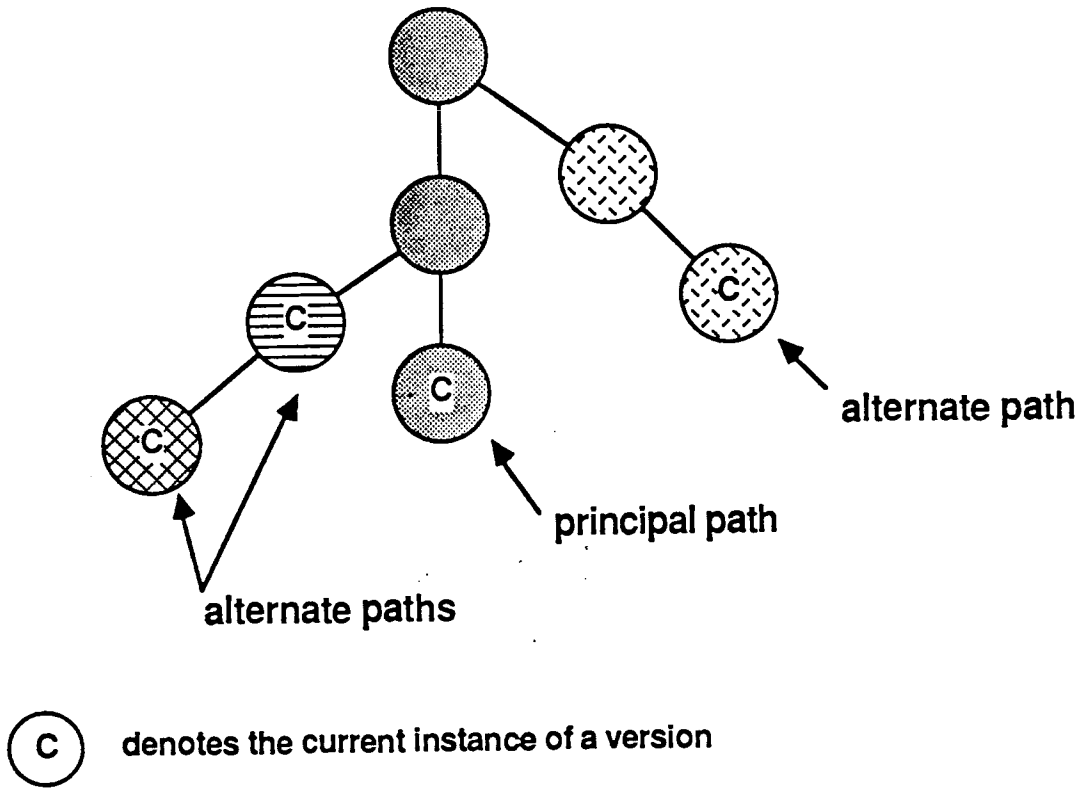


Figure III.4 - A Typical DHSS Object.

DHSS does not track any semantic relationships among the objects. Configuration management and the data model layers are responsible for monitoring and maintaining these relationships.

3.4.2 Object Naming and the Name Server Facility

The storage server supports a name mapping facility to associate an object name with a version of an object. When a DHSS object is created by a client that client must provide a unique name to be associated with the object. Each federation has a designated name server site. The function of the name server is to certify, within the federation, the uniqueness of each user-defined object name.

The storage server will map an object name first to a tree of instantiations and then to the particular representative in the tree that is the current version in the principal path. As alternate paths are created each path is assigned a number called its implicit alias. The implicit alias numbers are assigned within each tree in the order in which the alternate paths are created. Alternate paths may be named directly by the client or may be referenced by their object name and implicit alias. Non-current nodes are referenced by appending a time specification to a valid name. The time should be a time at which the desired instance was the current instance. Names without a time modifier are called floating references as the mapping function will map them to the current instance of some path and which instance is current changes over time. Names with time modifiers are pinned references. This naming scheme allows the client to specify any version in a tree of instantiations.

A valid object base name is of the following form.

Entity_Name	refers to the representative that is the current version of the entity.
Entity_Name[time]	refers to the version of the entity that was current at the specified time.

A valid Entity_Name is of the following forms.

Object_Name	refers to the representative that is the current version on the principal path of the object.
Alternate_Path_Name	refers to the representative that is the current version of an alternate path.
Object_Name(i)	where i is the implicit alias, refers to the current version of the i-th alternate path created in the object named.

A valid Object_Name or Alternate_Path_Name is an ordered concatenation of from one to three name parts separated by the symbol "^".

user_defined_name	the strings of symbols, excluding "^", as specified by a user.
user_name	the name of the user who specified the string.
site_name	the name of the site on which the string was specified.

Figure III.5 - Object Base Names for all Instantiations of a DHSS Object.

Figure III.5 summarizes the object base name references and the mapping performed by the storage server. Object names and path names are decomposable into three parts: a

user-defined name, a user name, and a site name. The goal of a flat name space is to ensure that the user-defined portion of a name is unique. A user should work under the assumption that the user-defined name is unique and can, for the most part, ignore the existence of the suffixed portions of the name.

3.4.3 Basic Access Capabilities in the Storage Model

DHSS provides basic access to the DHSS database through the checkout/checkin paradigm [32]. Execution of a checkout gives the client local copies of a set of instantiations. The end-user is then free to modify each copy in an appropriate manner (the tool program and the data model are responsible for monitoring and maintaining the semantics of the user's individual actions on the design data). Once the end-user's copy of the design has again reached a consistent state (as deemed by the data model), the client may checkin the updated design. Checkin adds a new current instance to the appropriate set of paths in a set of DHSS objects.

Checkout is a multi-reader request. Multi-reader is implemented by not setting a lock at the time of the checkout and providing each requesting client with a separate copy of the data. Checkin is a multi-writer request. By multi-writer we mean that every successful checkout may be followed by a checkin that will be accepted and saved by the storage system. If multiple checkouts have been performed on an object the first checkin executed will result in a successful update of the object. Subsequent checkins will result in the creation of alternate but related versions of the object. Figure III.6 shows (A) two clients performing a duplicate checkout of the current version of two DHSS objects, (B) the independent modification of the checked out data, and (C) the results produced by the checkin of the new versions.

For a user, the checkout-checkin sequence represents one user transaction. At the storage layer two transactions are executed, first a checkout (read) and then a checkin (write). This is consistent with Gray's definition of transaction, "A transaction instance is the unit of locking and recovery." [39]. The storage system does not hold locks on data while it is checked out, thus no storage system transaction is in progress during that time. DHSS supports the atomicity of storage layer transactions only. That is, either all results of

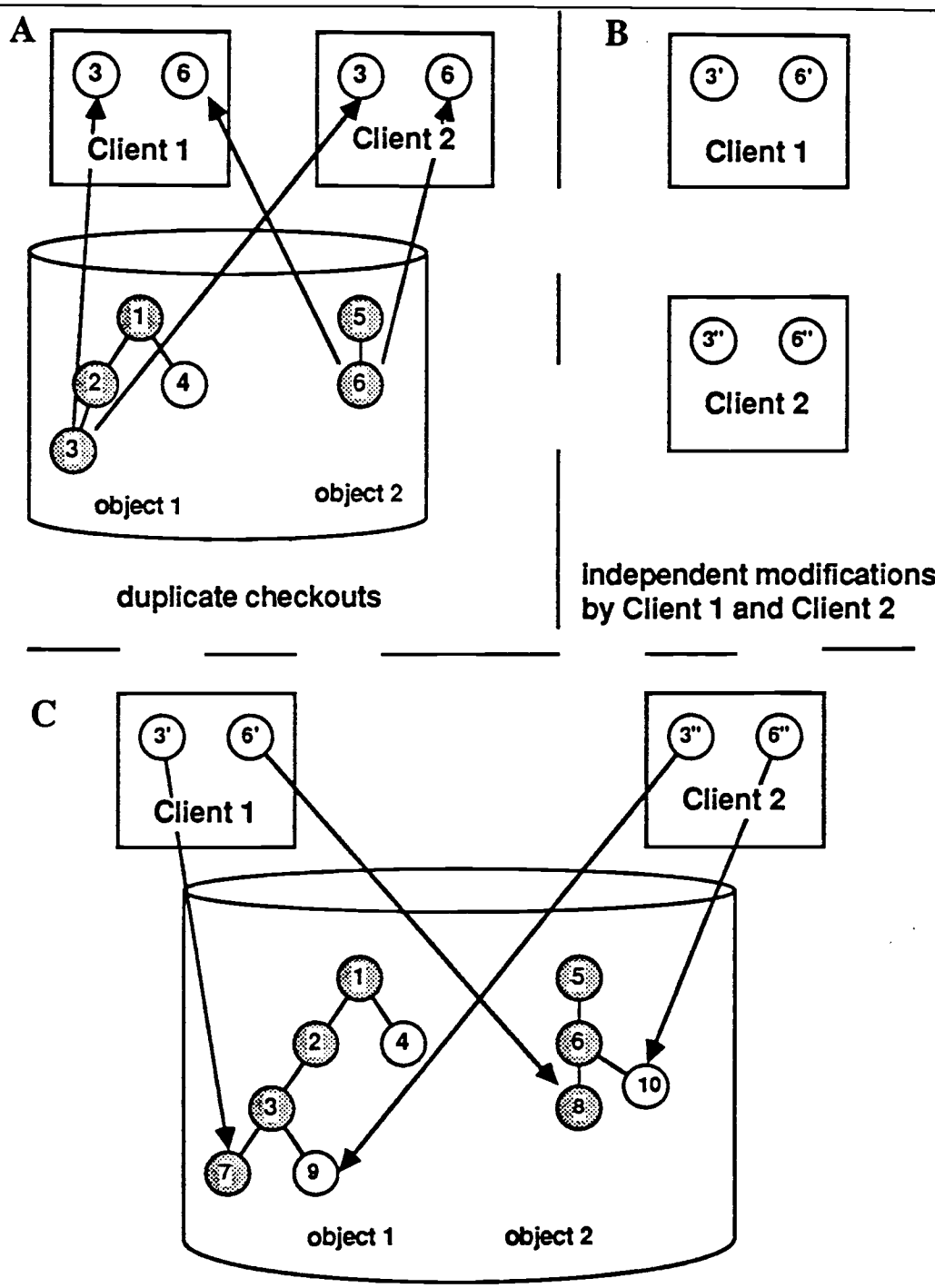


Figure III.6 - Multi-reader/Multi-writer Processing in DHSS.

the transaction will be in effect or none of the effects of the transaction will prevail. Support for the atomicity of user level transactions is handled by the layers above the storage system through the use of inverse storage system requests. A DHSS transaction will be referred to as a transaction or a request; a user level transaction will be referred to as a read-write transaction since it will typically contain both a read and a write action.

3.5 Data Manipulation in DHSS

DHSS supports a number of client requests for creating and manipulating DHSS objects. We assume a session interaction model where the client opens and closes existing federations and all intervening requests pertain to data in the object set of the open federation. Checkout, read_data, and return_data allow clients to look at objects. Create, update, and derive are used to make new versions of objects. Delete and erase remove instances. Assign and name manipulate the mapping of object names. Set_permission, grant, deny, new_owner, and usurp alter permissions that control user access to DHSS objects. Name_match queries the system directory to retrieve a list of object names. All requests except checkout, read_data, and return_data are considered to be write requests. Appendix I contains a detailed discussion of each request type supported by DHSS. In the following subsections we will discuss the requests of interest from the perspective of building a robust DHSS, these include the create request, the checkout request, the update request, the assign request, the delete request and the erase request. The message format for each request type is given using a standard notation for repetition where * means zero or more repetitions and + means one or more repetitions; groupings are enclosed in parentheses ( ); and optional terms are enclosed in square brackets [ ] [44].

3.5.1 Create

create object_name data_value

A single DHSS object is created by executing a create request. The request must specify a name for the object and an initial data value for the object. The proposed object name must be unique within the open federation. The name server site for the federation will approve or reject the name based on the grounds of uniqueness. If the object name is approved, the storage server will copy the initial data value to stable storage on the local site and add new entries for this object to the system directory. The requesting user will be the owner of the object. The newly created object will consist of a single instance that belongs to two paths, the principal path and the path having implicit alias one.

At creation time the new object is assigned an immutable token that is unique system wide. In DHSS the tokens consist of two parts the name of the site generating the token and a number unique to that site. The local site will be the primary site for the object. The primary site [1] is the control site for the synchronization of future actions on the object. The functionality of a primary site is discussed in detail in this chapter's subsection on transaction processing in DHSS.

3.5.2 Checkout

checkout existing_name^{+}

Execution of a checkout provides the client with a readable copy of the requested data. The checkout request must specify a set of valid names in the name space of the open federation. DHSS maps the set of names to a set of data instances and retrieves a readable copy of each of those instances. Information on who requested the checkout and what data was involved is maintained by DHSS. When data is acquired by a checkout, DHSS assumes that the data will be copied and modified by the client, and eventually stored in the database via an update request.

3.5.3 Update

update (existing_name new_data_value [replication_factor])+

The update request attempts to add a new temporal version to each specified path. Each update request must have been preceded by a checkout request for the set of paths to be updated. The set of new instance values is copied by the storage system to local stable storage. For each instance in the associated checkout, the primary site for the object containing that instance sets a lock on the object. This allows the updates to be serialized so that the first update based on instance X adds a new instance X' to the path and all subsequent updates based on instance X cause the creation of a new path with instance X as the root of that path. All late updates result in the implicit derivation of an alternate path followed by an update to the new alternate path. The client is notified of all late updates and the implicit alias numbers generated for the new paths.

An update is rejected if any instance in the specified path was not checked out or if the paths being updated were deleted or erased after the data was checked out. A failed update request has no effect on the user's local data copy. The local data may be used to create a new object.

3.5.4 Delete

delete object_name

An entire DHSS object can be removed by executing a delete request. The name of the entity specified in the delete request must be the name of an object rather than the name of a path. The name of a deleted object may not be immediately available for reuse as a unique name. Site or communication failures may require that the physical deletion of all references to the object be delayed. (See chapter six for a detailed discussion of deletions during failure states.) In any case, the object is logically deleted at the completion of the delete request.

3.5.5 Erase

erase name option

The erase request removes instances from paths. The two options for erasing are erase one and erase all. The semantics of an erase one request is to remove the current instance in a path reverting to the previous instance as the current. Erase one is implemented as an update of the instance being removed, updating it to be the previous version in that path. This approach insures that there are no time gaps between the temporal versions in a path. Thus all references to old versions dating back to the time of the creation of the path are well defined.

The erase all request will remove all instances in a single alternate path. The principal path cannot be erased. The name of the entity specified in the erase all request must be the name of a path. The name of the erased path may not be immediately available for reuse as a unique name due to postponed physical deletion. (See chapter six for details on deletion during failure states.) The implicit alias number corresponding to the erased path will never be reused.

3.5.6 Assign

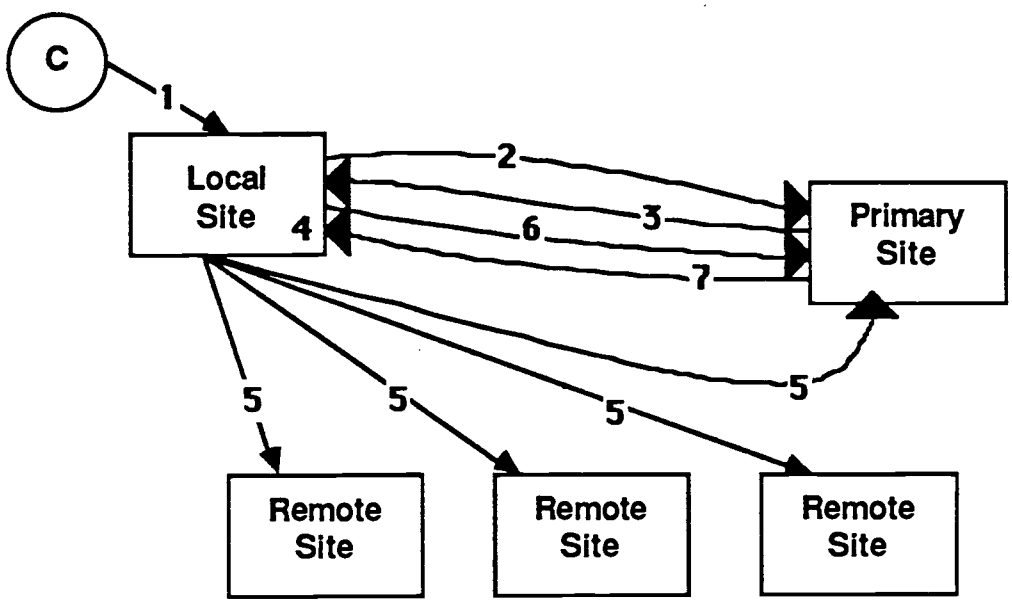
assign object_name path_name

Execution of the assign request alters the mapping of a DHSS object name in a manner contrary to the semantics of update. The assign request will force an object name to be mapped to the current instance of a specific path in the tree of instances. Semantically the assign allows the client to select a related alternate version of an object as the principal version of that object.

3.6 Transaction Processing in DHSS

The storage server is multi-threaded, that is, multiple requests are in a state of execution simultaneously. As defined by Bernstein et. al., an interleaved processing of requests is serializable [10, 12] if the results produced are equivalent to some sequential processing of the requests. DHSS produces a serializable execution of client write requests by employing primary site concurrency control [1]. Each DHSS object has an associated primary site. The primary site sequences the processing of requests on an object by functioning as a lock server for that object. If a request operates on object O, the requester must obtain a lock on O before the request can be serviced. Figure III.7 shows the steps required in processing a typical DHSS write request. The client's local site must obtain all required locks from the appropriate primary sites before the request can be processed. Once the locks are obtained the local site's directory should contain accurate information on the locked entities. The local site processes the request using the information in the local directory. When the processing is completed the results of the request must be incorporated into the directory of each site in the federation. This is accomplished by sending a multicast message to the subset of sites on the network that belong to the federation of interest. When the remote sites have acknowledged that their processing is complete, the initiating site will release all locks held for the processing of this request.

Primary site locking is a primitive concurrency control mechanism. Advantages of using primary site control are that the protocol is simple to implement and requires fewer messages to be exchanged than some other mechanisms. In applications where reading and writing are viewed as one transaction, primary site locking has some disadvantages: it greatly restricts the degree of interleaved execution of potentially conflicting requests [9], and site failures cause a discontinuity in service. In these traditional read-write transaction applications other concurrency control mechanisms such as timestamping [68], update tickets [52], two-phase distributed locking [68], voting quorums [36, 75] and optimistic protocols [50, 25] may be considered to be more desirable mechanisms based on their ability to support a higher degree of interleaved execution, to continue processing in the event of site or communication failure, to cause less time delay in the processing of requests, or to generate fewer network messages.



1. Receive request from local client
2. Request object lock from the primary site
3. Lock is granted
4. Process and commit the request locally
5. Multicast the result of the request (one-phase commit)
6. Request that the object lock be released
7. Lock release acknowledged

Figure III.7 - Processing a Write Request in DHSS.

Because DHSS is a multi-reader/multi-writer system separate read and write transactions are supported; this nullifies many of the benefits derived from the other concurrency control mechanisms. When a client makes a write request, the only reasons the request may fail to be processed are: the client did not have permission to execute this request; the object or path name(s) specified in the request have been deleted or are unknown by the system; or the client did not make the requests in a valid order (i.e. a checkout must always precede an update). All of these factors can be checked by the client's local site before performing any real processing. Once these tests have succeeded, the request will always be processed; thus there is no reason to use a voting scheme concurrency control mechanism to serialize this write request. What remains to be done is to sequence the write

requests so that they will be processed uniformly at all sites. This sequencing is easily accomplished by a primary site.

All DHSS requests are processed atomically. This means that the results produced by a single transaction are committed to the database before they are seen by any other transaction. Processing of a multicast message is also performed atomically. Atomicity of execution is achieved by establishing for each transaction a separate (lightweight) task that holds all results until the transaction reaches its commit point. Once the commit point has been reached the task forces all results to the disk in one synchronous write. A synchronous write is a blocking event for the entire storage system, thus no storage system processing can proceed before the write completes.

The normal processing protocol defined by this seven step algorithm maintains the internal consistency and mutual consistency of the database.

> **Proposition 1:** In a failure free environment, normal request processing maintains internal consistency at a single site.

Proof of Proposition 1: In a failure free environment, where site crash, communication failure, software failure, and media failure do not occur, all DHSS requests are well-defined, that is, each request type takes one internally consistent database state to another internally consistent database state. The storage server is structured to execute requests atomically. We will enumerate all requests processed by a site, ordering them by the time of their local commit point. Such an enumeration exists because the set of local commit points must be serial. Given that the initial database state is consistent, assume that the database state after processing (N-1) requests is consistent. By induction, the processing of an N-th request leaves the database in a consistent state. The N-th request reads a consistent database state because the (N-1) requests committed previously left the database in a consistent state. Given that each request type is well-defined, the N-th request takes a consistent database to a consistent database. Thus, normal processing in a failure free environment maintains internal consistency at a single site.

Proposition 2: In a failure free environment, normal request processing maintains mutual consistency among all sites in a federation.

Proof of Proposition 2: We consider the seven step algorithm for processing a write request in DHSS. Let site S_1 be the site of a client request R. In processing steps one through four, site S_1 processes R and commits the results of R locally. In processing step five, all remote sites in the federation are informed of the request R and R is processed at each of the remote sites. For each site S_i, we can construct an enumeration E_i of all requests processed by that site. The enumeration E_i reflects the serial ordering of each request's local commit point at site S_i. Request R will appear in each enumeration. The seven step processing algorithm maintains mutual consistency among the site within the federation if (a) all enumerations E_i would consist of the same set of requests should request activity cease, and (b) for every request C that conflicts with request R either $C \rightarrow R$ (read as "C precedes R") in every enumeration E_i, or $R \rightarrow C$ in every enumeration E_i.

(a) Given a failure free environment, all sites receive multicast notification of all processed requests, thus all enumerations E_i converge to the same set of requests.

(b) If C is a request that conflicts with R, then the write set of C intersects the write set of R. Let X be the data entity in the intersection of the write set of C with the write set of R. In step two of the processing algorithm, request C must obtain a lock on X. Similarly, request R must obtain a lock on X in processing step two. The primary site associated with X will grant the lock to one request; the other request will be suspended until the lock can be obtained. Suppose request R is granted the lock and request C must wait. The processing sequence for request R in a failure free environment would proceed as follows. Request R is processed locally by site S_1; request R is multicast to all sites in the federation; all remote sites execute and commit request R; request R is added to the enumeration of requests at each site; site S_1 releases the lock held on X; and the primary site grants the lock on X to request C. Clearly $R \rightarrow C$ in every enumeration E_i. If request C were granted the lock prior to request R then $C \rightarrow R$ in every enumeration E_i.

Every enumeration E_i converges to the same set of requests and all conflicting requests participate in the same $\rightarrow$ relationship on every site; therefore, the DHSS processing algorithm maintains mutual consistency among the sites within a federation.

In a failure free environment, this primary site and one-phase commit protocol ensures the internal consistency and mutual consistency of all sites in the federation. Given that communication or site failures may occur at any step in the processing sequence, we must augment this simple protocol to maintain internal consistency and mutual consistency in the faulty environment we have defined. In chapters four, five, and six we present the robustness protocols to complement this transaction processing algorithm.

3.7 Group Transactions

Providing a checkout and checkin facility for single objects is not sufficient support for the upper layers of a CAE database system. The application tool layer and the representation layer work with complex objects which may be decomposed into many sub-objects. Each sub-object is stored as a single DHSS object. These upper layers will request data in terms of complex objects. The configuration management layer will decompose a complex object request into its many subcomponents in order to provide the higher layer with a configuration of the complex object. A configuration is a version of a complex object and is constructed by combining a version of each subcomponent of the complex object. The configuration specification is stored as an object which refers the sub-objects. A configuration may make fixed references or floating references to the sub-objects. Fixed references always map to the same version of a sub-object. Floating references map to the version that is current at the time the reference is processed. When a configuration contains floating references, the storage system must provide a checkout and checkin facility for groups of objects in order to provide the user with a consistent version of the complex object being operated on. Storage system requests that operate atomically on a group of DHSS objects are called group transactions.

3.7.1 Consistency in Group Checkout

The checkout and update transactions of DHSS operate on a group of data items from one or more DHSS objects. The group of instances are the subcomponents in a configuration. When a group checkout is executed, DHSS records the status of each item as either current of a principal path, current of an alternate path, or non-current. Use of a group checkout insures the client of configuration read consistency.

Definition: A set of instances is configuration read consistent if

1. the set of instances is specified as a configuration by the configuration management layer;
2. in the configuration, all instances named by floating references will be simultaneously current versions.

If a group checkout requesting current instances of a set of objects were carried out as a sequence of singleton checkouts, property 2 of configuration read consistency could be violated. If the local site processes an update affecting those instances yet to be read the final result of the sequence of reads will not produce simultaneously current versions of the instances. Thus, group checkout is necessary to provide the client with a consistent set of data values.

Execution of a group checkout is complicated only in that the local site may have to request data copies from several distinct remote sites. Neither local locking nor remote locking is required in the processing of a group checkout because all transactions are executed atomically. This ensures that the local site successfully maps the names in a group checkout to a configuration consistent set of instances.

3.7.2 Consistency in Group Update

A client may execute a group update only on a set of instances obtained by a group checkout. Group update must maintain the configuration write consistency of the instances being updated. We will define a set of rules for processing group updates. Adherence to these rules will maintain configuration write consistency among the items in a group update. The intention of the update rules is to form an alternate configuration object when a principal configuration cannot be updated consistently. A modified definition of configuration write consistency and a modified set of update rules may be desired for maintaining configuration write consistency when modeling other applications such as computer-aided software engineering.

For a CAE application, we propose the following definition of configuration write consistency for a set of instances.

Definition: A set of instances is configuration write consistent if

1. the set of instances is specified as a configuration by the configuration management layer;
2. every instance updated will be current at the same time;
3. every temporal version of the configuration is completely contained in temporal versions of principal paths or is completely contained in temporal versions of alternate paths.

Based on this definition, we have constructed a set of group update rules to maintain configuration write consistency. Application of the update rules requires knowledge of the currency status of each instance at the time of the update and the currency status of each instance at the time of the checkout. The update rules for configuration consistency in DHSS are as follows [32].

1. If all instances are the current version of a principal path then apply the updates to the principal paths producing new temporal versions of each object in the group.

2. If all instances are the current version of an alternate path then apply the updates to the alternate paths producing new temporal version of each alternate path in the group.

3. If some subgroup of the instances are the current version of an alternate path and the other instances are non-current versions of their respective paths, then for each of the non-current instances create and substitute a new alternate path rooted at that instance and carry out the updates according to rule 2.

4. If some, but not all, of the instances are the current versions of a principal path, then for each instance that is the current of a principal path create and substitute a new alternate path rooted at that instance and carry out the updates according to rule 3.

Group update processing in DHSS, as defined by the group update rules, maintains configuration write consistency.

Proposition 3: In a failure free environment, normal request processing maintains configuration write consistency.

Proof of Proposition 3: Group update processing by DHSS maintains configuration write consistency if (a) the set of entities updated form a configuration, (b) the set of new instances are current at the same time, and (c) all new instances are added to principal paths or all new instance are added to alternate paths.

(a) Based on configuration management we will assume that any group checkout and its corresponding group update represents a modification of a configuration, thus the set of entities updated represent a configuration.

(b) We know that DHSS processes all requests atomically; therefore, all new instances added by a group update become current simultaneously.

(c) The site local to the group update request must process the request according to the rules of group update. All other sites will be informed of the group update request by a multicast message containing the results of the request. The remote sites will not participate in any decision making, thus, in satisfying criteria (c), it is sufficient to show that the group update rules applied by the local site will force the new instances produced by a single group update to be added to all principal paths or all alternate paths. Group update rule 1 clearly adds all new instances to principal paths only.

Group update rule 2 adds all new instances to alternate paths only. Group update rule 3 dictates that non-current updates be placed in new alternate paths and all current updates must be applied to alternate paths. Clearly rule 3 adds all new instances to alternate paths only. Group update rule 4 dictates that all current updates to principal paths must be placed in alternate paths and all current updates to alternate paths will be applied to those alternate paths. Clearly rule 4 adds all new instances to alternate paths only. Thus, the group update rules force all new instances created by a group update to be added to principal paths only or to alternate paths only. In a failure free environment, the group update rules must be applied as defined; therefore, normal request processing in a failure free environment maintains configuration write consistency in DHSS.

3.7.3 Locking during Group Update

Execution of a group update requires that a lock be obtained from the appropriate primary site for each entity in the group. Thus steps two and three of figure III.7 must be repeated until all necessary locks have been obtained. Similarly steps six and seven of figure III.7 must be repeated until all the acquired locks have been released. When multiple locks are requested by multiple transactions deadlock may occur. If a lock is currently held by a transaction T_i and transaction T_j requests the lock, transaction T_j enters a transaction_wait_for relationship with transaction T_i. Transaction T_i in turn may be in a transaction_wait_for relationship with another transaction. If a cycle exists in the transaction_wait_for relationship a deadlock has occurred. Gray [39] presents three ways to deal with this problem: timeouts, deadlock avoidance, or deadlock detection.

A timeout scheme assigns a time limit to each transaction. The transaction must acquire its locks within the time limit or be backed out and restarted. The main disadvantage to this approach is that as system load increases the number of transactions that exceed their time limit increases even though deadlock may not have occurred. The backing out and restart of each transaction adds to the already increasing system loading tending to make the problem worse.

Deadlock avoidance can be used in applications that meet these two requirements.

1. Every transaction processed by the system can predeclare the complete read set and write set for the transaction before requesting any reads or writes.

2. There exists a total ordering on all data items stored by the system.

The protocol for deadlock avoidance is that all transactions must request their locks in increasing order based on a total ordering imposed on the data. This guarantees only acyclic chains of transactions in every transaction_wait_for relationship.

Deadlock detection is the most general approach to the problem of deadlock. Several graph based algorithms have been proposed for the detection of deadlocks [37, 39, 59]. Gray proposed a global deadlock detector process executing at only one site. All sites maintain a local transaction_wait_for graph to detect local deadlocks and send all local information to the global detector. The global deadlock detector constructs a global transaction_wait_for graph and attempts to enumerate cycles. If cycles are found the deadlock is broken by selecting, from each cycle, one transaction to be backed out and restarted.

In DHSS we use a deadlock avoidance protocol. The two requirements for using deadlock avoidance are satisfied by DHSS.

1. Every request declares its read set prior to executing any reads and the write set is contained in the read set or introduces completely new data items.

2. Each DHSS object has an associated immutable token. The tokens are totally ordered thus the set of DHSS objects is totally ordered.

3.7.4 Processing a Group Update

All DHSS transactions request locks on DHSS objects in increasing order of the tokens associated with those objects. When all locks have been granted for a group update, the initiating site compares the status of the entities at the time of the checkout with their current status and processes them according to the four group update rules given above. If new alternate paths must be derived the local site does so by requesting a new implicit alias

number from the primary site for the DHSS object and adding a new path to the tree. The entire tree of versions are held by the lock, thus no new locks need to be obtained in order to derive new alternate paths. When all the required new paths have been derived, the updates are performed on the new set of updatable paths. Once the update has been committed locally, the results of the update are multicast to all the sites in the effected federation. A lock release request is sent to the appropriate primary site for each DHSS object involved in the update. After all the processing steps of figure III.7 have been completed the requesting user can be informed of the outcome of the request. With the addition of the robustness mechanisms we will see that it is reasonable to inform the requesting user of the result of the request immediately after committing the results of the request on the local site. This can greatly decrease the apparent response time for a group request because the user does not have to wait for the multicast messages and the lock release messages to be sent and acknowledged.

3.8 Other Version Servers

The concept of a server storing temporal versions and alternate versions of data is not a new one. Those in the area of software development have been aware that the maintaining of versions was critical in supporting the process of designing and implementing software. The early work in CAE database systems also maintains that versions are an intrinsic part of the design environment.

3.8.1 Operating System Utilities for Version Control

Operating systems normally supply utilities for version control of files. Examples of such utilities are the Source Code Control System (SCCS) [67] and the Revision Control System (RCS) [76] which run under the Berkeley Standard UNIX operating systems. These utilities support a multi-reader and a single-writer checkout and checkin facility. The version files reside on a single machine and the user must know exactly where the version files are stored on that machine. The user must insert "ID keywords" into the data itself;

these keywords are used by the version control system to determine what data corresponds to a particular version. The creation of alternate versions is done explicitly by the user.

SCCS and RCS both support a merge of temporal versions in a single path. RCS will even attempt to perform a merge of alternate versions. Such merge operations do not make use of any semantic information about the data stored in the files. This often results in the merging of logically inconsistent information. Clearly this type of version control system is designed specifically for storing text strings and is to be used directly by an end-user.

3.8.2 A Version Server for CAE

Katz, Anwarrudin, and Chang propose a version server for CAE applications [48]. Their server includes pieces of functionality we have placed in the configuration layer, the data model layer, and the polymorphic representation layer of our global system architecture. The data stored by the version server is untyped but structural relationships among the data items are tracked by the server. Data is classified in three planes: the version plane relates temporal and alternate versions (which they call derivatives and alternatives respectively), the configuration plane relates those instances that comprise a version of a higher level object, and the equivalence plane relates different physical representations of the same entity.

Multi-reader and single-writer access is supported. Execution of a checkout provides the user with a copy of the data in the user's private workspace. Checkins are two level operations. When modified data is checked into the system, the data is moved to a semi-public workspace where consistency checks are performed. If the modifications reflected in the new version are consistent with other components in the database and revisions by other users, then the modified version is installed in the database.

Each version tree has a single name associated with it. The name maps to a particular instance in the version tree. This instance is the root of some subtree and only instances in that subtree may be updated. An assign operator is used to move the name to another instance thus moving the updatable subtree of instances.

The version server is currently being prototyped on a network of VAX-11[8] machines and SUN[9] workstations. How failures would effect the multi-reader and single-writer model were not discussed; also, recovery and robustness mechanisms for the version server have not been investigated.

3.8.3 The DOSS Version Server

Weiss, Rotzell, Rhyne, and Goldfein have proposed the Design Objects' Storage System (DOSS) for storage of design data [81]. DOSS stores temporal and alternate versions of objects and supports multi-reader and single-writer access to those objects. The system is distributed in that any site having sufficient storage space stores copies of the data and functions as a server. The server sites maintain fully redundant directories on the data stored by the system. The non-server sites may cache information on the location of data items they have referenced.

Every decomposable object has an associated immutable object-id (OID). To checkout data to a private workspace, users specify a list of OIDs. If the local site is not a server and has no cached information on an OID, then the local site broadcasts a message asking the location of the desired OID.

The robustness of the system is addressed only from the perspective of permanently removing machines from the network and migrating the data stored on them to other sites. The processing of checkins during normal processing or failure modes was not investigated.

[8] VAX-11 is a trademark of Digital Equipment Corporation.

[9] SUN is a trademark of SUN Microsystems.

CHAPTER FOUR
ROBUST DATA ACCESS IN THE DISTRIBUTED STORAGE SYSTEM

Our goal is to make DHSS an optimistic robust distributed storage system with respect to the failure classes of site crash, network partition, and media failure.

Definition: An optimistic robust distributed storage system is a system that, even when system components have failed, will successfully and optimistically process transactions, will preserve internal consistency at every site, will preserve mutual consistency among groups of communicating sites, and will preserve configuration write consistency among groups of communicating sites.

Our approach to achieving optimistic robustness consists of two parts;

1. a set of rules and mechanisms to support maximum access to user data
2. a set of recovery protocols for maintaining internal consistency, mutual consistency, and configuration write consistency among groups of communicating sites.

An overview of our approach, the set of rules and mechanisms to support maximum access, and recovery protocols for a storage system running in a single local area network was presented in [30]. In this chapter we will define rules for data access and present mechanisms for supporting those rules in DHSS. In the next two chapters we will present recovery protocols for maintaining consistency in DHSS.

4.1 Optimistic Access in DHSS

In order to maximize user access when sites or communications have failed, DHSS supports the following access rule for the processing of transactions.

> Optimistic Access Rule: Given a running site and an authorized user, an entity may be checked out if a copy of all parts of the entity can be acquired. Any entity that has been previously checked out may be updated. The requests assign, name, delete, erase, derive, set_permission, grant, or deny may be processed if they affect DHSS objects that are known to exist. Requests to create new DHSS objects are always processed.

Under the optimistic access rule almost all requests submitted at running sites during the failure of other components of the distributed system are executed. Requests that cannot be processed include: a checkout request for data that is not accessible due to a failure; and an update request on data that cannot be checked out. Requests such as assign, name, delete, erase, derive, set_permission, grant and deny can be performed on an object O only if the site processing the request knows that object O exists. If object O does exist, information about O is contained in the site's system directory.

Unlike distributed database systems built for other applications, DHSS can support robust optimistic access control because under normal processing conditions DHSS ignores the possibility of update conflicts until they actually occur. If a failure partitions the network, DHSS may unknowingly process conflicting updates in separate partitions. In some sense the system ignores these conflicts until the failure is repaired and the partitions are brought back together. When the partitions are merged, the conflicting updates "occur" and can be managed, as they would be in a failure free environment, by implicitly deriving alternate versions. The general philosophy is that the results of conflicting requests will be managed by merging the results of the requests or by allowing one result to prevail over another.

If DHSS is to support the optimistic access rule, we must continue to process requests under most every possible circumstance. Processing a DHSS request may require a combination of four types of service:

1. approval of a name by the federation's name server site,
2. locking of a DHSS object by the primary site associated with that object,
3. generation of a unique implicit alias number by the primary site associated with the affected DHSS object,
4. providing another site with a copy of user data stored by the local site.

If we wish to continue processing requests when failures occur, the services provided by sites other than the local site must be performed by a replacement site. Service support must be capable of migrating among the sites belonging to a federation. Services responsibilities are migrated from one site to another by the use of pseudo name server sites, pseudo primary sites, and multiple copy sites. A pseudo name server site is one site that acts as a temporary replacement for the federation's true name server site. A pseudo primary site for a DHSS object is one site that acts as a temporary replacement for the object's true primary site. If a site holds a copy of a data item then that site is a copy site for the data item. To provide checkout capability during a failure, each data item will have multiple copy sites.

Placing multiple copies of each instance of data on distinct sites is done whether or not the distributed system has suffered any component failures. In contrast, designating and using a pseudo name server site or a pseudo primary site is an action that is taken only when a failure is noted by some site in the system. This implies that we must utilize some mechanism for keeping track of failures noted by active sites. We use a modified version of the virtual partition protocol [33] described in chapter two of this thesis.

First we present a modified virtual partition protocol for tracking site and communication failures. Next we present a protocol for selecting pseudo name server sites and pseudo primary sites. Finally we present our approach to data replication and the placement of redundant data copies.

4.2 A Modified Virtual Partition Protocol

We use a modification of the virtual partition protocol defined by El Abbadi, Cristian, and Skeen [33] to track possible failures and the correction of possible failures in our environment. A virtual partition is a group of sites in one federation that have agreed that they will talk only to each other for the processing of transactions. Sites belong to different virtual partitions for each federation to which the site belongs. If failures did not occur and no new sites ever enrolled in the federation, the virtual partition would consist of all sites belonging to the federation. A management protocol allows the list of sites belonging to a virtual partition to be modified. Detection of a possible failure or detection of the correction of a possible failure signals a need to modify the list of sites in the virtual partition. The virtual partition will lose sites from the group when a failure occurs and add sites to the group when a failure is corrected.

Every virtual partition has a name. The partition names are generated such that each name is unique, the set of all partition names generated forms an ordered set, and the names are generated in an increasing sequence. In DHSS, virtual partition names are decomposable into two parts. The first part is a count that reflects the number of times the virtual partition membership has been modified. We refer to this count as the level number for the virtual partition. The second part is the name of the site that initiated the virtual partition. When a federation is defined by a client on a site S, the name of the first virtual partition for that federation is 1S and site S is the only member of partition 1S. When a request is processed by the sites in a virtual partition the results written by that request are tagged with the name of the partition. This tag represents the fact that the sites belonging to the virtual partition named must have been informed of the result.

A virtual partition is an attempt to form mutually exclusive groups of sites that have two-way communication with each other. A list of the sites belonging to a virtual partition may not be correct in that sites which are not reachable may be in the list and sites that are reachable may not be in the list. The site list for a virtual partition may require modification if any site in the virtual partition receives a message from a site not in the virtual partition, or if any site in the virtual partition fails to receive a response from a site belonging to the virtual partition. If a site S believes that the list of sites belonging to the virtual partition is incorrect, then the site S must initiate a new virtual partition.

```
Initiate a New Virtual Partition

Phase One:
    Construct a new virtual partition name
    Construct an invitation containing the new partition name and a list of
            all sites in the federation
    New Members = { }
    Send the invitation to all known federation sites, including the local site
    Begin a timeout on the arrival of each reply
    For each site invited Do
        If a reply is received
        Then Add the responding site's name to the set of New Members
    EndFor
Phase Two:
    Construct and send a "commit" message containing the new
            virtual partition name and the list of New Member sites
            to the set of New Member sites
```

Figure IV.1 - Algorithm for Initiating a New Virtual Partition.

Formation of a new virtual partition is achieved by a two-phase protocol. Figure IV.1 presents the algorithm for initiating a new virtual partition. In phase one the initiator site S first constructs a new virtual partition name by adding two to the current virtual partition's level number or the largest level number proposed in this federation, and appending the site name S. (The reason for adding two, rather than incrementing by one, is explained in the chapter five.) Next site X sends a message to all known sites in the federation inviting them to join a new virtual partition. The message contains the new proposed virtual partition name and a list of all the sites believed to be members of the federation.

The sites receiving the invitation examine the proposed virtual partition name and the list of sites. Figure IV.2 presents the algorithm for accepting or rejecting virtual partition invitations. A receiving site accepts the invitation if the proposed virtual partition name is greater than the name of the partition it currently belongs to or of a partition name it has accepted, and if the list of sites contained in the invitation includes all sites believed to be members of the federation. If the invitation is acceptable, the receiving site sends an acceptance message to the initiating site. An acceptance message contains the name of the site and the virtual partition history of that site. A virtual partition history (VPH) is a list of the names of all of the virtual partitions a site has been a member of. A receiving site rejects the invitation if the proposed virtual partition name is too small or if any sites that are known to be members of the federation are missing from the list of sites contained in the invitation.

```
Processing Virtual Partition Invitations

If you are in the process of joining a new virtual partition
Then Case of phase in the partition join process
        Phase One:
            If the new proposed name is < the older proposed name
            Then
                Do not reply to this invitation
                Continue processing the previous invitation
                Exit
            EndIf
        Phase Two:
            Continue processing the previous invitation
            Exit
    EndCase of
EndIf
If the new proposed name > the current or older proposed virtual partition name
Then
    If the invitation's site list contains sites not known by the local site
    Then
        Add the new sites to the local site's list of sites for this federation
    EndIf
    If the invitation's site list is missing sites known by the local site
    Then
        Do not reply to this invitation
        Begin to initiate a new virtual partition
    Else
        Suspend processing in this federation
        Send an acceptance message containing the local
                virtual partition history to the proposing site
        Begin a timeout on the arrival of a commit message for this invitation
    EndIf
Else
    If you are not currently proposing a new partition
    Then
        Do not reply to this invitation
        Begin to initiate a new virtual partition
    EndIf
EndIf
```

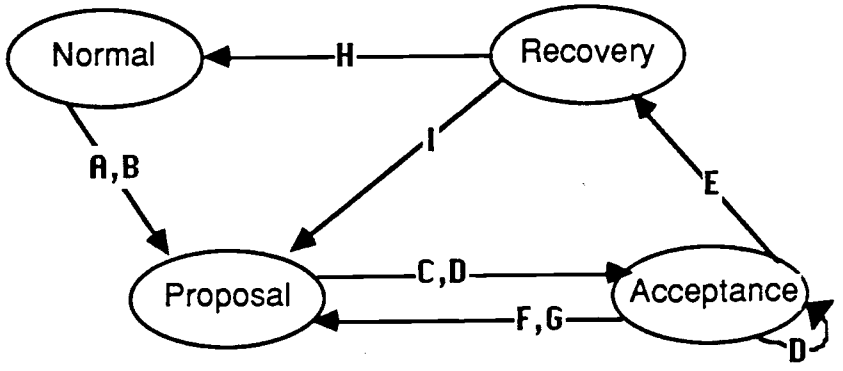
Figure IV.2 - Algorithm for Processing Invitations to Join a New Virtual Partition.

If the invitation is rejected and the receiving site has not already accepted a better invitation, then the receiving site must initiate of a new virtual partition.

The initiating site associates a timeout interval with each invitation sent. If the time interval elapses without receiving a reply from the site then that site will not be included in the new virtual partition. When each invitation has either timed out or been replied to, the initiating site enters phase two of the protocol and sends a commit message to those sites

that replied. The commit message contains the name of the new virtual partition, each unique virtual partition history, and the list of sites that reported that partition history.

Each site that received an invitation sets a timeout on the commit message associated with that invitation. If the commit message does not arrive within the time limit, the receiving site will attempt to initiate a new virtual partition. If the commit message arrives within the time limit, the receiving site enters phase two of the partition protocol.



Conditions:

A - Receive a message from a site outside of the current virtual partition
B - Fail to receive a reply or response from a remote site within a specified time limit
C - Accept your local site proposal
D - Receive a better proposal
E - Receive the commit message
F - Fail to receive the commit message within a specified time limit
G - Receive a proposal with an unacceptable proposed partition name but announces the existence of new sites in the federation
H - The recovery is completed and the site commits to a new virtual partition
I - Fail to receive a message from a committed site within a specified time limit

Figure IV.3 - States and State Transitions of a Running Site.

If a site is running, it must be in one of four states: normal, proposal, acceptance, or recovery. Transitions among these states are controlled by the virtual partition negotiation protocol. Figure IV.3 presents the possible transitions and the conditions that cause those transitions.

The negotiation protocol will generally converge and return all sites to the normal state. There are environments in which the protocol will cause all sites to cycle between the proposal and acceptance states or the proposal, acceptance, and recovery states.

> **Proposition 4:** There exists one or more environments in which the virtual partition negotiation protocol will cause all sites to cycle among the recovery states never returning to a normal processing state.

Proof of Proposition 4: We construct such an environment as follows. Let F be a federation residing on two sites S_1 and S_2. Sites S_1 and S_2 are running and executing requests. The following sequence of events leads to cyclic recovery.

1) S_1 is slow in processing messages.
2) S_2 believes S_1 has failed.
3) S_2 proposes a new virtual partition.
4) S_1 sends an acceptance message to S_2's proposal.
5) S_2 is slow in processing messages.
6) S_1 times out the commit message expected from S_2.
7) S_1 proposes a new virtual partition with a larger name.
8) S_2 accepts the proposal due to the larger name.
9) S_1 is slow in processing messages.
10) S_2 times out the commit message expected from S_1.
11) S_2 proposes a new virtual partition with a larger name.
12) Go To Step 4.

A similar scenario can be constructed where the sites S_1 and S_2 cycle from the proposal state to the acceptance state to the recovery state. Therefore, there exists one or more environments in which the virtual partition negotiation protocol exhibits cyclic behavior.

We note that in our constructed environment the cyclic behavior is dependent on sites S_1 and S_2 cycling from fast to slow to fast in opposition. The site is said to be fast if the total load on local system is such that a message can be acknowledged within a specified time limit; otherwise the site is said to be slow. If either site were to be slow in responding to two consecutive messages, then the faster site would timeout the commit message, timeout the reply to the new proposal, commit to being a single site partition, and resume service for the local clients. The probability of this cyclically opposing behavior continuing

indefinitely, goes to zero as time goes to infinity. We can encourage an early break in the cycle by allowing each site to select random timeouts greater than some minimal timeout T. This should add sufficient variability to the timeout behavior of the sites avoiding cyclic failures.

When the sites eventually enter the recovery state, each site determines what type of recovery is taking place and executes an appropriate consistency protocol that will ensure that the local site is mutually consistent with the other sites in the new virtual partition. Once mutual consistency has been achieved, each site officially commits itself to being a member of the new virtual partition. Committing to membership in a virtual partition means that the site remembers the name of the new virtual partition, remembers the list of sites belonging to the new virtual partition, and uses the new virtual partition name as the tag value for the results of each write request.

Two recovery protocols will be employeed to ensure the mutual consistency of the sites in a new virtual partition. If all of the sites in the new virtual partition reported the same virtual partition history then the virtual partition will perform a divergence recovery. If any site reports a disparate virtual partition history then the virtual partition will perform a merge recovery. The details of how mutual consistency is achieved during a divergence recovery and a merge recovery are discussed in the following chapters.

4.3 Pseudo Name Server Sites

To support optimistic access in DHSS, we use pseudo name server sites as a means of providing name server functionality in each virtual partition. A pseudo name server site is a substitute for the true federation name server site. When a federation is defined, the site at which the define request was submitted becomes the true name server site for the federation. When processing transactions, communication is limited to those sites in the current virtual partition. If the true name server site is not in the current virtual partition then a pseudo name server site must be selected.

The selection process requires the existence of a total ordering on the set of sites. At least two natural total orderings exist: order the sites alphabetically by site name; or order

the sites by the network address associated with each site. If the true name server site is not in the current virtual partition then the smallest site, according to the ordering on sites, will become the pseudo name server for this virtual partition. This protocol ensures that each virtual partition contains exactly one site acting as the name server for the partition.

Lemma 1: Each virtual partition contains exactly one name server site.

Proof of Lemma 1: Let V be a virtual partition in federation F. Let site N be the true name server site for federation F.

Case 1: Suppose virtual partition V contains site N. There exists exactly one true name server site; therefore, all sites in V recognize one unique name server site in V.

Case 2: Suppose virtual partition V does not contain site N. A pseudo name server site is selected. Each site in V selects the minimum site in V based on some ordering of the sites. By definition of the selection protocol, the ordering attribute produces a total ordering over all sites. The partition V is a subset of all sites; therefore, the sites in V are totally ordered. Each site uses the same ordering rule and the minimum of a totally ordered set is a single site; therefore, each site in V selects the same unique site as the pseudo name server site for partition V.

Each name server site maintains a flat name space, within its respective virtual partition, by not allowing sites within that virtual partition to create duplicate object base names. It is possible that sites in one virtual partition will propose object base names that are unique within that virtual partition but are duplicates of object base names proposed and approved as unique in another virtual partition. When a merge recovery takes place these object base names will appear as duplicates. Such duplication cannot be allowed. Duplicate object base names are avoided in such situations by appending modifiers to each duplicate name. The natural modifier that will make an object base name unique is the name of the site on which the creating request was submitted. This suffix may be more or less meaningful to the client who is attempting to disambiguate an object base name. A more meaningful suffix is the name of the user that created the object base name. Unfortunately, the name of the user may not be sufficient. A user may be forgetful or malicious and create two objects with the same name at two distinct sites that can not communicate with each other. For these reasons, we use as a suffix: first the name of the user that created the object base name and second (if necessary) the name of the site on which the creating request was submitted.

When an object base name is created, the name of the creating user and the name of the site on which the creating request was submitted are saved with the user-defined portion of the name. When requests are processed by the storage server, each object base name specified in the request is looked up in the system directory. If multiple directory entries are found for the user-defined portion of the name, the user receives a reply that the name is ambiguous. The user may then query the storage system using the name_match request to obtain a list of fully modified names with matching prefixes. The user may repeat the original command using a modified name which is unique. It is important to note that if the user-defined portion of a name is unique over the set of known object base names, then the normal reference using only the user-defined portion of the object base name is sufficient.

Appending modifiers to the user-defined portion of a name to produce unique object base names could be used in place of name server sites as a general mode of operation. We prefer the use of a name server site and pseudo name server sites because they maintain a closer approximation to a truly flat name space. Modified names should be the exception rather than the rule. Users who subscribe to this philosophy can use the name request to rename the user-defined portion of a modified name, making the name unique and flattening the object base name space.

4.4 Pseudo Primary Sites

To support optimistic access in DHSS, we use pseudo primary sites as a means of providing primary site functionality for each accessible object in each virtual partition. Primary sites are responsible for granting locks on objects, thus guaranteeing sequenced updates to those objects. The true primary site is the site on which the DHSS object was created. When a write request is processed by the storage system, each DHSS object affected by the request must be locked. Request processing is always limited to those sites in the current virtual partition. If one or more of the required primary sites are not in the current virtual partition then each of the missing primary sites must be replaced by a pseudo primary site. A pseudo primary site substitutes for the true primary site by granting locks on behalf of the true primary site. When a site other than the true primary site receives a request for a lock, that site must check to see that it has been elected as a pseudo primary site

for the data item being locked. If the site is not the pseudo primary site an appropriate reply is returned to the site that requested the lock. The requesting site may seek to become the pseudo primary site.

Algorithm for Nominating the Local Site

```
If the local site has agreed to the nomination of another site, during the current
        partition, as the pseudo primary site for data item X in federation F
Then Start a timeout on the commit message for this nomination
Else
Phase One:
    Construct a multicast nomination message for the local site and
            data item X in federation F
    The local site votes "YES" on its own nomination
    Send the nomination message to all sites in the current virtual partition
    Begin a timeout on the receipt of a reply from each site
    For each reply Do
        Case of Reply
            "YES": Continue
            "NO": Begin timeout on receipt of a better nomination for X in F
            Timeout: Initiate a new virtual partition
        EndCase
    EndFor
    If all votes are "YES"
    Then
    Phase Two:
        Multicast a commit message for the local site as pseudo primary for
                data item X in federation F
        If all acknowledgements are received
        Then The local site is elected
        Else Initiate a new virtual partition
    Else If the time limit on receipt of a better nomination expires
        Then Initiate a new virtual partition
    EndIf
EndIf
If the time limit on receipt of a commit message has expired
Then Initiate a new virtual partition
```

Figure IV.4 - Algorithm for Nominating a Pseudo Primary Site.

The process of selecting a pseudo primary site is based on a two phase election protocol. If a client, local to a site S, requests a write operation on a data item X, then site S must obtain a lock from the primary site for X before processing the request. If the true primary site for X is not in the current virtual partition of site S, then site S must determine whether or not a pseudo primary site for X exists in the current virtual partition. A pseudo

primary site for X exists if the local site believes that a pseudo primary site has been elected, no merge recoveries have been performed since the election completed, and the elected pseudo primary site is in the current virtual partition. If neither the true primary nor a pseudo primary site for X is in the current virtual partition, then site S will initiate a two phase election of a pseudo primary site. Figure IV.4 presents the algorithm for nominating a pseudo primary site. In phase one, site S will nominate itself as the pseudo primary site and all sites in the current partition will cast a vote. If elected, in phase two site S will send a commit message to all sites in the current partition, receive an acknowledgement from each site in the current partition, and then consider itself to be an elected pseudo primary site. This two phase protocol is similar to the asynchronous-timeout-based negotiation protocol for forming a virtual partition.

Figure IV.5 presents the algorithm used by the voting sites. When a site receives an election proposal, that site must send a reply accepting or rejecting the nomination. A site will reply with a rejection only if it has previously received a nomination for a site that is "larger" than the newly proposed site according to the total order on sites or it has already commited, during the current virtual partition, to a different pseudo primary site. After accepting one nomination a site not yet committed to that nomination may accept a better nomination. Eventually some "largest" site that desires to be the pseudo primary site for data item X will be elected. If a failure or timeout occurs during the election, a new virtual partition must be formed. Recovery and pseudo primary site election are interwoven in that recovery will terminate pseudo primary site elections. When the recovery is completed, the request that required the lock must re-evaluate the accessibility of a primary site in the new current virtual partition. If a primary site is not a member of the new virtual partition, then the election process will be initiated again.

Once a pseudo primary site is elected that site is used as the primary site until the pseudo primary site is no longer in the current virtual partition or a merge recovery is performed. When a merge recovery is performed all pseudo primary sites must relinquish their position. Elected pseudo primary sites continue to function in any new virtual partition formed by a divergence recovery. In a divergence recovery, we are ensured that the true primary site could not have rejoined our virtual partition. If all sites in the larger partition agreed on site S as the pseudo primary site for X, then all sites in the new reduced partition hold the same agreement; thus, we may continue to use the elected pseudo primary site if that site remains in our virtual partition. When a merge recovery is performed, the sites in

```
Algorithm for Processing a Nomination Message

Phase One:
    Receive a message nominating site S as a pseudo primary site for
            data item X in federation F
    If the local site has committed to a different pseudo primary site for X
            during the current virtual partition
    Then Reply "NO"
    If the local site has voted "YES" on a previous nomination during the
            current virtual partition, but has not committed to it
    Then
        If S is ≥ previously nominated site
        Then
            If the previously nominated site is the local site
            Then Begin a timeout on the commit message
            Reply "YES"
        Else Reply "NO"
        EndIf
    Else Reply "YES"
    EndIf
Phase Two:
    If the timeout on the commit message expires
    Then Initiate a new virtual partition
```

Figure IV.5 - Algorithm for Electing a Pseudo Primary Site.

each of the merging partitions may have different pseudo primary sites for X; therefore, pseudo primary sites must relinquish their status following a merge recovery.

> **Lemma 2:** The pseudo primary site election protocol converges and elects exactly one site in a virtual partition.

Proof of Lemma 2: Let $S = \{ s_1, s_2, ..., s_k \}$ be the set of sites that simultaneously nominate themselves as a pseudo primary site for data item X in federation F.

Case I: Assume that no site or communication failures occur during the election process. Each site s_i sends a nomination message to all sites in the current virtual partition and receives a nomination message from every other site in S. The sites s_i will accept each increasing nomination. By the total ordering on the set of all sites, there exists a largest site s_L in the set S. All sites will accept the nomination of s_L. Site s_L will receive "YES" votes from all sites because s_L is the largest site in S. The site s_L is the only site elected if every other site in S receives at least one "NO" vote. By the protocol, each site in S sends a nomination message only if no other nomination has been received already; therefore, each site in S sends and accepts its own nomination before receiving any other nominations. Site s_L has accepted its own

nomination when sending the nomination message to the other sites; therefore, site s_L sends a "NO" vote to every other site in S. Therefore, the protocol converges and site s_L is the only site elected by the protocol.

Case II: Assume that a site or communication failure occurs during the two phase pseudo primary election protocol. The detection of a possible failure by one of the nominating sites or one of the voting sites causes the termination of the election and the negotiation of a new virtual partition. When the recovery is completed, a new election may be performed. We claim that the previous terminated election has no effect on the new election. Once a recovery has completed, the set of initiating sites nominate themselves as the pseudo primary site if no other site in the new partition has done so. Therefore, the terminated election has no affect on the nominating sites. Each voting site casts its vote based on the nominations and commitments made during the current virtual partition. It is the case that a voting site gives a positive reply to the first nomination for data item X in each virtual partition. The terminated election has no affect on the voting behavior of the voting sites in the newly formed virtual partition because the terminated election was performed in the previous partition.

Therefore, a terminated election has no affect on an election performed in a different virtual partition. If K consecutive elections are terminated due to failures, then the (K+1)st election is not affected by the previous K terminated elections and by case I the (K+1)st election converges and exactly one site is elected.

> **Corollary 1:** Each virtual partition contains at most one functional primary site for each data item.

Proof of Corollary 1: Let V be a virtual partition in a federation F. Let X be a data item in federation F.

Case 1: Suppose virtual partition V contains the true primary site for data item X. There exists exactly one true primary site for each data item and every site in partition V agrees on which site is the true primary site for X; therefore, partition V contains exactly one primary site for data item X.

Case 2: Suppose virtual partition V does not contain the true primary site for data item X. If there is an existing pseudo primary site S for X in partition V then that pseudo primary site was elected using the pseudo primary site election protocol. By lemma 2, the election protocol elects exactly one site in partition V. If there is no existing

pseudo primary site S for X in partition V or a site believed to be a pseudo primary site replies that it is not the pseudo primary site, then a pseudo primary site election will be performed. By lemma 2, this election protocol will result in the election of exactly one pseudo primary site for X in partition V.
Therefore, all sites in V recognize exactly one site S as the pseudo primary site for X.

Corollary 2: Within a virtual partition, at most one request may hold a lock on a data item at any given time.

Proof of Corollary 2: Let site P be the primary site or the pseudo primary site for a data item X in a virtual partition V. By the defined locking protocol, site P will allow only one lock to be set on X at any time. By corollary 1, we know that P is the only primary site for X in partition V. Therefore at most one request will hold a lock on X at any given time.

4.5 Data Replication

To support optimistic access in DHSS, it is imperative that we maintain multiple copies of each item of user data. In most systems, data replication enhances only read access. In DHSS, data replication enhances both read access and write access because read access is the only necessary condition for write access. Our problem of data replication can be divided into two parts: how many copies should be made of a data item and where should those copies be placed. When the physical environment for the distributed system is a single local area network, the two subproblems are treated independently. When the physical environment is a sequence of local area networks connected by gateway sites, the solution to the two subproblems will dependent on how the federation sites are distributed across the physical environment. We will examine the subproblems of quantity and placement independently before considering the more complex environment in which they interact.

4.5.1 Determining a Replication Factor

The create request and the update request cause new data to be stored in the database. In each of these commands the user may optionally specify a replication factor for the new data item. A replication factor of one may be specified. In this case the system will make only one copy of the data item even though this greatly increases the probability that the data could become inaccessible for reading and consequently inaccessible for updating. If no replication factor is specified a default replication factor is used. Each federation is assigned a default replication factor when the federation is defined; either the user defining the federation specifies a factor to be used as the federation wide default or the DHSS system wide default factor (currently two) is used.

4.5.2 Placement of Redundant Data

When a create request or an update request is processed, the first copy created is stored on the site at which the request was issued and processed. When the results of the request are multicast to all sites in the current virtual partition the multicast message also contains a list of the sites that should store a redundant copy of the data. When such a multicast message is received and processed the site that issued the message is recorded as holding a copy of the data item; all other sites listed are recorded as designated but not yet confirmed copy sites. If a site finds its name among the list of designated copy sites then that site must attempt to obtain a copy of the data from some site that is known to have a copy. Once a site has obtained a redundant copy of a data item that site will multicast this fact to all sites in the current partition thus altering its status from a designated copy site to a confirmed copy site.

The algorithm for constructing the list of N designated copy sites is presented in detail in figure IV.6. The main factors to be considered in selecting placement sites are which sites are designated as fileserver sites for the federation, which sites are in the current virtual partition, which sites store previous versions of the data item, which users are permitted access to the data, and how much storage space has already been contributed by each site. If at least (N-1) fileserver sites are in the current virtual partition, then select (N-1) of the

```
Selecting N Designated Copy Sites

If N >= the number of known sites in the federation
Then
        Select the set of all known sites in the federation
Else
        Unselected sites = { all known sites in the federation minus the local site }
        Selected sites = { the local site }

        For each site in the set of unselected sites Do
                Set this site's preference rating to zero
        EndFor

        For each site in the set of unselected sites Do
                If this site is a fileserver site
                Then Add 3 to this site's preference rating
                If this site is a member of the current virtual partition
                Then Add 3 to this site's preference rating
                If the data item being replicated is NOT the first instance
                                in a new path or a new object
                Then If this site stores a previous temporal version of the new data item
                        Then Add 2 to this site's preference rating
                If this site has local users that are permitted access to this
                                data item by private inclusion
                Then Add 1 to this site's preference rating
        EndFor

        Order the sites having preference ratings > 0 by decreasing rating
        Order the sites having preference ratings = 0 by increasing space used
        Append the list ordered by space used to the end of the first list
        Add the first (N-1) sites in the total list to the set of selected sites
        Return the set of selected sites
EndIf
```

Figure IV.6 - Algorithm for Selecting Designated Copy Sites.

fileserver sites as designated copy sites giving preference to those fileserver sites that stored the previous temporal version of this data item or have contributed the least storage space to storing DHSS objects. If the current partition does not contain sufficient fileserver sites, then select other sites giving preference to those sites that stored the previous temporal version of this data item or have contributed the least storage space to storing DHSS objects.

The selection algorithm may be forced to select sites that are not members of the current virtual partition. When a merge recovery is performed these sites will learn of their selection as designated copy sites. After the recovery is completed these designated sites will request a data copy from a site that is known to store a copy. Once a copy is obtained, the site will multicast this fact to all sites in the current virtual partition, thus changing its status from a designated copy site to a confirmed copy site.

In the case of a create request or an update request that adds a new path, the selection algorithm gives preference to fileserver sites and those sites that have contributed the least amount of storage space. For an update to an existing path, preference is given to sites that store the previous temporal versions of that data item. These sites are preferred because we can make use of the ancestral relationship of data instances to reduce the storage space required to store multiple versions of an object.

If a site stores two temporal versions of a data item, the most recent version will be stored as a full copy and the older version can be stored as a backward difference based on the newer version. A difference [70] is conceptually an errata list specifying the differences between an old version and a new version of a data item. A backward difference is a list of changes that must be applied to the newer version to produce the older version. If a site acquires a full copy of a new data instance I and the site has a full copy of a previous temporal version O of this same instance, the old full copy O should be replaced with a backward difference with respect to the new copy I. The instances O and I participate in a differenced_from relationship for the local site; instance O is differenced_from instance I. Backward differences are used rather than forward differences so that we may store full copies of all current instances. This should save time when processing checkouts because most checkout requests are for current instances which are always stored as full copies.

If a user requests a checkout of an older version and that version is stored as a difference on site S, then the storage server on site S must reconstruct the desired version. The storage system directory contains information on the differenced_from relationship for the local site so we know how to reconstruct older versions. Reconstruction of an older version is accomplished by following the differenced_from relationship forward to locate a full copy and then backwards, applying the changes stored in each difference in the order specified by the reverse of the differenced_from relationship.

Figure IV.7 shows three consecutive temporal versions from one path of a DHSS object and a possible placement of those versions on three sites S_1, S_2, and S_3 assuming a replication factor of two. An update creates a new path instantiated by instance 1 and places a copy of instance 1 on sites S_1 and S_2. A checkout and update are performed on instance 1 to create instance 2 and copies are placed on sites S_1 and S_3. Another checkout and update are performed on instance 2 to create instance 3 and copies are placed on sites S_2 and S_3. The local representation of each instance and the differenced_from relationship are recorded

Copy Placement	Three Instances
Sites S_1 and S_2	
Sites S_1 and S_3	
Sites S_2 and S_3	

Create Version 1

	version	format	differenced_from
Site S_1	1	full	null
Site S_2	1	full	null
Site S_3	1	none	null

Update Produces Version 2

	version	format	differenced_from
Site S_1	1	diff	2
	2	full	null
Site S_2	1	full	null
	2	none	null
Site S_3	1	none	null
	2	full	null

Update Produces Version 3

	version	format	differenced_from
Site S_1	1	diff	2
	2	full	null
	3	none	null
Site S_2	1	diff	3
	2	none	null
	3	full	null
Site S_3	1	none	null
	2	diff	3
	3	full	null

Figure IV.7 - Placement of Full and Differenced Copies of Three Instances on Three Sites.

in the system directory at each site. If the storage server at site S_1 is asked to provide a copy of instance 1, the directory at S_1 states that instance 1 is stored as a difference and that it was differenced from instance 2. Since instance 2 is stored as a full copy at S_1, the storage server applies the changes for instance 1 to the full copy of instance 2 producing a full copy of instance 1.

Leblang and McLean [53] claim that the average number of changes made between one version and the next will result in a backward difference that uses between 1 and 2 per cent of the storage space required by the full version. Using backward differences results in a large reduction in the total space required to store all versions of an entity. The only restriction we have placed on differencing is that the two temporal versions must be from the same path in a DHSS object. This restriction is not necessary, but was adopted in an attempt to minimize the storage space used. Two instances from distinct paths in a DHSS object or from two distinct DHSS objects are logically unrelated to each other. Differencing two logically unrelated instances is likely to result in differences that are larger than 5 per cent of the the size of a full copy.

4.5.3 The Effects of Site Distribution on Data Replication

Our previous discussion of data replication presented replication factors and data placement as independent problems. We proposed a scheme of static or user specified replication factors and data placement based on dynamic factors known by the storage system. The goal of data replication is to increase the probability that data will be accessible when failures occur. It is possible to better serve this goal by studying replication factors and data placement as dependent problems. We propose an algorithm that dynamically calculates replication factors, and an algorithm for data placement based on static and dynamic characteristic of the system and its environment.

In chapter three we discussed the concept of a probable partition. Probable partitions are those physical partitions that result from the crash of a gateway site. Given a specific network configuration, all of the probable partitions can be calculated. This information can facilitate a better placement of data copies. If users in multiple probable partitions are permitted access to a data item, then each probable partition should store a copy of that data.

This strategy not only decreases the probability that a data copy will become unavailable due to a failure, but also decreases the response time for obtaining data copies by ensuring that a copy is located within the local area network. The number of redundant copies to be placed is dynamically calculated as the number of probable partitions containing sites that service users who are permitted to access the data item being replicated.

This strategy may not be as reasonable as it first appears. Placing one copy of a data item in each probable partition in which it can be accessed is defeated if the copies are placed at the gateway site. The probable partitions are formed by the failure of the gateway site; when the gateway site fails, all copies at that site become inaccessible. If we avoid storing copies at gateway sites we lose the use of a large amount of storage space. Even worse, gateway sites are often fileservers for the LANs they are connected to.

There are two environments in which this strategy works well. Some LANs have multiple fileserver sites with only one of the sites serving as a gateway. The non-gateway site could be used as the fileserver in that LAN. Some LANs have one or more workstations sites with a large disk or an optical disk. Such workstations serve as fileservers and are rarely used as gateway sites. When we have one of these environments in a large number of the LANs we prefer to dynamically place data copies by probable partition access. If the majority of the LANs in the environment do not possess these properties we use our previous approach of static or user-defined replication factors and data placement based on preference ratings.

CHAPTER FIVE
MAINTAINING CONSISTENCY IN THE DISTRIBUTED STORAGE SYSTEM

In the previous chapter we presented optimistic access as part one of our two part approach to making DHSS an optimistic robust distributed storage system. The second requirement for achieving robustness is a set of recovery protocols to augment normal transaction processing. These protocols maintain internal consistency, mutual consistency, and configuration write consistency among groups of communicating sites. A virtual partition represents a group of communicating sites, thus we must preserve consistency within each virtual partition. By proposition 1, proposition 2, and proposition 3, (presented in chapter three) we demonstrated that in a failure free environment, normal request processing is sufficient for maintaining internal consistency, mutual consistency, and configuration write consistency. A properly functioning virtual partition is a failure free environment.

> **Corollary 3:** In the failure free environment of a single virtual partition, normal request processing maintains internal consistency at each site, mutual consistency among the set of sites in a virtual partition, and configuration write consistency for update processing.

Proof of Corollary 3: By proposition 1, proposition 2, and proposition 3, we know that in a failure free environment, normal request processing maintains internal consistency at a site, mutual consistency among all sites in a federation, and configuration write consistency for update processing. A single virtual partition presents a failure free environment for a subset of the sets in a federation. The sites within the virtual partition will carry out normal processing among exactly those sites in the virtual partition. Therefore, within the failure free environment of a single virtual partition, normal request processing maintains internal consistency at a site, mutual consistency among the sites in the virtual partition, and configuration write consistency for update processing.

When membership in one virtual partition must be abandoned for membership in another virtual partition the assumption of a failure free environment has been violated. The normal request processing protocol is insufficient for maintaining consistency when transitioning from one virtual partition to another. Virtual partitions reconfigure when sites

cannot be reached or when sites considered to be unreachable begin communicating. We will now present the protocols for maintaining consistency in the event of diverging virtual partitions and merging virtual partitions.

5.1 Divergence Recovery

One virtual partition splits into two or more virtual partitions when a site crashes or a communication failure occurs. First the failure must be detected. Next a new virtual partition is formed, and finally any inconsistencies among the sites in the new virtual partition must be corrected. We will now define the protocol for each of these steps.

5.1.1 Detection of a Failure

We say that a site A has detected a failure if any site B in the current virtual partition fails to reply to a service request within some time limit, or fails to acknowledge a multicast message within some time limit. If a site fails to reply to a service request, we must reconfigure the virtual partition so that a pseudo service site can be selected and the client's request can be processed. If a site fails to acknowledge the processing of a multicast message, a mutual inconsistency may exist between the site that sent the multicast message and the receiving site. Any site that believes it has detected a failure must initiate a new virtual partition.

It is important to note that the detected failure may not be a true failure. In either case, the failed site B may or may not be a member of the next virtual partition depending on its response to the virtual partition invitation. If site B has not crashed and is not partitioned from the other sites in the virtual partition, the recovery is still necessary to ensure mutual consistency among the sites in the newly formed virtual partition. If site B is not a member of the newly formed virtual partition it is impossible to determine whether site B has crashed, is partitioned due to communication failure, or is excessively slow in responding to messages.

5.1.2 Current Inconsistencies

A failed site may cause a current inconsistency among the surviving sites. The source of this inconsistency is seen by studying the processing of requests on a site.

A site processes requests by obtaining locks from primary sites, performing local processing of the request, and multicasting the results of the request to all other sites in the virtual partition. Few network facilities provide true multicast communications that are reliable. True multicast communication is the sending of a single message to a selected set of sites by placing the message on the network exactly once. Reliable multicast would ensure that either all of the selected sites received the message or none of the selected sites received the message. In general, multicast communication is simulated by point-to-point communication between the sender and each receiver of the message. This means that the single message must be sent repeatedly, once to each destination site. If a site was in the process of multicasting a message by a sequence of point-to-point messages and the site crashes, then some of the destination sites will receive the message and some will not receive the message. This results in an inconsistency among the surviving sites.

Clearly, part of the solution to this problem is to require the sites that did receive the failed site's last message to propagate that message to those sites that did not receive the last message. We implement such a propagation strategy by placing restrictions on the sending of multicast messages, by logging each multicast message, and by requiring some synchronization during the processing of a request.

We will restrict the sending of multicast messages such that each site may send at most one multicast message at any time. Requests that have completed their local processing must queue for the multicasting of their results. Each site will log the most recent multicast message received from every site including itself. This requires dedicated log space for exactly one multicast message per site in the federation.

Each multicast message will contain a sequence number. Multicast message sequence numbers are generated in increasing order by each site for the multicast messages they send. The multicast message sequence number will be logged by each receiving site as part of the multicast message. These sequence numbers are part of the information which will be passed during the two phase negotiation of a new virtual partition. Figure V.1 shows the

1. The Virtual Partition Initiator sends an Invitation containing:
   Proposed Virtual Partition Name
   List of all sites known to be members of the federation

2. A Site receives the invitation, and sends a reply containing:
   Local site name
   The virtual partition membership history for the local site
   For each site in the current virtual partition, one pair containing:
      site name
      sequence number of the last multicast message logged for this site

3. The Virtual Partition Initiator sends a Commit message containing:
   Name of the new virtual partition
   For each unique virtual partition history reported, a list containing:
      The virtual partition history
      A list of the sites that reported this history
      For each site reported by a site with this history, one triple containing:
         site name
         multicast message sequence number
         name of a holding site that reported this sequence number

Figure V.1 - Information Exchanged During Virtual Partition Negotiation.

information that must be exchanged by the virtual partition initiator and the members of the virtual partition during the two-phase virtual partition negotiation.

During virtual partition negotiations, each site receiving an invitation will include, in the acceptance message, multicast sequence number information for each site in its current virtual partition. The information will be a list of pairs: <site name, sequence number contained in the last multicast message received from this site>. The virtual partition initiator will process the pairs. If more than one sequence number is reported for a site S by two sites in the same virtual partition, then an inconsistency exists. Because a site can be in the process of sending at most one multicast message at any time and remote sites must acknowledge processing each multicast message, there may be at most two distinct sequence numbers reported by sites that were members of the same virtual partition. That is, a site may miss at most one multicast message before a virtual partition negotiation and recovery is executed. For each site reported in a pair, the initiator will add an ordered triple to the commit message of the new virtual partition. Each ordered triple contains: <site name, largest sequence number reported, name of site reporting this sequence number>. The virtual partition initiator will multicast the commit message to those sites that agreed to join the new partition. When the commit message is received, each site will compare the

sequence number in each triple with the appropriate message in the multicast message log. If the sequence number in the log is smaller than the sequence number in specified in the commit message a copy of the missed multicast message must be acquired from the site specified in the triple. When the multicast message is received it is logged in the multicast message log and processed to completion. This process is performed for each triple in the multicast message. If multiple multicast messages were missed and had to be acquired they may be processed in any order because they represent the results of non-conflicting requests.

Lemma 3: The set of messages propagated by the message propagation phase of a divergence recovery represents the results of non-conflicting requests.

Proof of Lemma 3: Let S be a site performing divergence recovery from partition P_1 to partition P_2. Let M be the set of multicast messages that require propagation to site S. Suppose M contains two messages m_1 and m_2 that represent conflicting requests R_1 and R_2 respectively. R_1 and R_2 were processed and committed in the virtual partition P_1. If R_1 and R_2 conflict, then there exists a data item X such that X is contained in the write set of R_1 and X is contained in the write set of R_2. Normal request processing requires that every request obtain a lock on each item in its write set, thus, R_1 must hold a lock on X and R_2 must hold a lock on X. This is a contradiction by corollary 1. Therefore, the set of multicast messages to be propagated during a divergence recovery represent a set of non-conflicting requests.

When all multicast sequence numbers have been compared and all missing messages obtained and processed, the site commits to membership in the new virtual partition. Thus divergence recovery is a three-phase protocol: the two-phase virtual partition negotiation and the propagation of missed multicast messages. Figure V.2(A) presents the algorithm used by the virtual partition initiator to process the sequence number pairs reported by all joining sites. Figure V.2(B) presents the algorithm for message propagation as performed by all members of the newly formed partition.

(A) - Algorithm for processing replies to a virtual partition invitation.

```
For each reply received Do
    Locate the sequence number info for the partition history in this reply
    For each pair <A, N> in this reply Do
        Locate the information triple for the site A specified in the pair
        If the triple is not found or
                the triple's sequence number ≠ reported sequence number
            Add the reporting site's name S to the pair
            Add this new triple <A, N, S> to the info for this partition history
        EndIf
    EndFor
EndFor

For each distinct partition history Do
    For each site A reported in any triple <A, N, S>
            associated with this partition history Do
        Add to the commit message the triple having the largest value of N
    EndFor
EndFor
```

(B) - Algorithm for processing sequence numbers in divergence recovery.

```
For each triple <A, N, S> in the commit message Do
    If the sequence number N is > the multicast message sequence number
            in the local multicast message log for site A
    Then
        Request a copy of the missing multicast message from site S
        Log the multicast message
        Process the multicast message
    EndIf
EndFor
```

Figure V.2 - Algorithms for Processing Sequence Numbers.

This message propagation protocol is effective in that all sites remaining in a virtual partition formed by a divergence recovery will eventually process the same set of requests.

Lemma 4: After a sequence of one or more divergence recoveries, all sites belonging to the newest partition will eventually have processed the same set of requests.

Proof of Lemma 4: Let S be the set of sites that have performed a sequence of divergence recoveries to achieve membership in virtual partitions $P_1, P_2, \ldots P_n$. By the definition of a divergence recovery, we know that for each divergence recovery from partition P_i to partition P_{i+1}, the sites belonging to partition P_{i+1} were also members of partition P_i, and there exists at least one site in partition P_i that did not receive an acknowledgement of a multicast message or could not communicate with a service site in partition P_i. All such sites in partition P_i will attempt to initiate a new virtual partition P_{i+1}. When a site initiates a new virtual partition, all processing at that site is suspended for the federation undergoing the recovery. When a site receives an invitation to join a new virtual partition, all processing in the affected federation is suspended until a recovery is completed. When processing is suspended, each site may divide the requests it was processing into four set: (a) requests whose processing is completed, (b) at most one request whose results are being multicast to the other sites, (c) requests that have committed locally and are waiting to have their results multicast to the other sites, and (d) requests that have not yet committed locally.

(a) Requests of type (a) have been processed by all sites in partition P_i; therefore, all sites in partition P_{i+1} have processed all type (a) requests.

(b) Requests of type (b) have been processed by some subset of the sites in partition P_i. For each type (b) request R_j, let M_j be the set of sites that have processed request R_j in partition P_i. The set of sites M_j is non-empty because M_j must contain the site that initiated R_j. If any site in M_j is a member of the new partition P_{i+1}, then the multicast message associated with R_j will be propagated among the sites joining partition P_{i+1}. Thus, if any site in partition P_{i+1} has processed a type (b) request, then all sites in partition P_{i+1} will receive and process that request.

(c) and (d) Requests of type (c) and type (d) will become requests of type (a) or (b) with respect to some future partition P_k where $k \geq (i+1)$. By our previous arguments for requests of type (a) or type (b), these requests will be processed by all sites belonging to a future partition P_k.

Therefore, all sites in a partition created by a divergence recovery will eventually process the same set of requests.

5.1.3 Preemption of Requests

When a divergence recovery is completed, the sites remaining in the newly formed partition will process new requests in the failure free environment of the new virtual partition. From proposition 1, proposition 2, and proposition 3, we know that normal request processing in a failure free environment is sufficient to maintain internal, mutual, and configuration write consistency. These results assume that the system has been failure free from time zero when the storage system was empty and zero requests had been processed. In particular it assumes that no requests are in the state of being processed when the failure free environment begins. This assumption does not hold for the environment at time zero of the new virtual partition. Requests that are being executed at the time recovery processing begins are suspended. These suspended requests will either be resumed or backed out and restarted. If all requests were backed out and restarted, the system processing state could be discarded and we would achieve the pristine state assumed by propositions one through three. Divergence recovery does not require that all requests in progress at the initiation of recovery be backed out and restarted. We selectively perform backout and restart on requests that may cause a loss of mutual consistency in the new partition and requests that may cause hung requests in the new partition. A hung request is one that is blocked by a request that is no longer active in the current virtual partition.

In the next two sections we will discuss why requests may be hung during future request processing, how the divergence recovery protocol releases these requests, why mutual consistency may be lost during future processing, and how the divergence recovery protocol will indirectly manage future consistency.

5.1.3.1 Avoiding Hung Requests

When a divergence recovery is completed, the remaining sites should be able to process almost all requests. The only restriction is that the sites must know of the existence of the data and a copy of the data must reside on at least one site in the new partition. The ability to continue processing after a recovery may be hindered by side effects of requests that were being processed by sites excluded from the new virtual partition. In particular,

requests that are partially processed by an excluded site may have set locks at primary sites. The primary sites may be members of the new virtual partition. Because the requesting site has been excluded from the new partition these resources will not be released by the requesting site. If a new request requires a lock that is held by a partially processed request on an excluded site, the new request would be hung.

Definition: A request R is hung if R is blocked by a sequence of requests and one of the requests in the sequence is no longer active in the current virtual partition.

To avoid hung requests, every site in the new virtual partition must release locks that are held by requests being executed on sites which are now excluded from the new partition. As part of divergence recovery, each primary site must scan the local lock table discarding all entries requested by sites that are not members of the new partition. If a request R has its locks released for this reason, then from the perspective of the newly formed partition, R has been preempted.

Definition: A request R is preempted if all locks currently held by R are released, R is backed out, and R is restarted.

The locks granted to request R are released by one partition. We will see in the next subsection that the backout and restart of request R is performed in another partition that has as a member the site which initiated R.

Lemma 5: In the time interval between the completion of a divergence recovery and a site or communication failure in a federation F, there will be no hung requests in F.

Proof of Lemma 5: Assume that following a divergence recovery forming virtual partition P in Federation F and prior to a site or communication failure affecting F, there exists one or more hung requests in federation F. Let h be a hung request in F. By the definition of a hung request, from among the requests executing in federation F there exists one or more requests that are no longer executing in partition P and h is blocked by such a request. Let r be the request that is not executing in partition P and is blocking h. The divergence recovery protocol preempts all requests that are initiated by a site lost when partition P was formed; therefore, request r must have been initiated after the divergence recovery forming partition P. Then request r was initiated by a site in partition P but request h is hung so request r must be executing

outside partition P. Request r can be initiated by a site in partition P and be executed outside of partition P only if the initiating site has failed. By our assumption that there are no site or communication failures affecting F, this is a contradiction.

Pending name reservations at a name server site are managed in the same manner. If the site that requested the pre-reservation of a name is not a member of a new partition formed by a divergence recovery then the name server site will release the name reservation. The pre-reserved name may be reused by any site remaining in the partition following the recovery.

5.1.3.2 Avoiding Future Loss of Mutual Consistency

Transitioning from one partition to another by a divergence recovery can result in a loss of mutual consistency during normal processing in the newly formed partition. Consider the following sequence of events for three sites S_1, S_2, and S_3 in partition P_1. Site S_1, executing a request R_1, is granted a lock on data item X by primary site S_3. Site S_2 believes that site S_3 is not reachable and initiates a new virtual partition. The processing of request R_1 is suspended on site S_1 during the divergence recovery. A new partition P_2 is formed consisting of sites S_1 and S_2. On site S_2 a new request R_2 requires a lock on data item X. Because site S_3 is not in the current partition of site S_2 the pseudo primary site election algorithm is used to elect site S_2 as the pseudo primary site for data item X. Request R_2 will request and be granted a lock on data item X by site S_2. Request R_1 and request R_2 both hold a lock on data item X. This behavior violates the mutual exclusion requirement of the locking protocol and may lead to a loss of mutual consistency.

This loss of mutual consistency can be avoided by preempting request R_1. All requests that have been granted locks by sites that have been excluded from the new partition must be preempted. The divergence recovery protocol must backout all such requests so they may be restarted in the newly formed virtual partition.

5.1.4 Preparing for Future Merge Recoveries

A failed site may cause future inconsistencies among the sites in a virtual partition. If the failed site was detected by one or more sites sending a multicast message of their latest results, these results would have an associated virtual partition tag value that was the name of the pre-failure virtual partition. The purpose of the tag value is to signify that all sites belonging to that virtual partition have knowledge of those results. This constraint may be violated when a site fails. In particular, the failed site will not have knowledge of the results specified in a multicast message that caused the discovery of the site failure. These erroneous tag values will cause a future inconsistency, because there is no way to discover that the failed site does not have knowledge of these particular results.

The solution to this problem is to retag the results specified in each of the logged multicast messages, using a future virtual partition name as the tag value. This has the effect of pushing the results forward in partitioned time. As we will see in the section on merge recovery, this establishes the state required for merge recovery to locate and process all information that may be unknown to a site which is attempting to join a virtual partition.

Each site participating in the divergence recovery must scan the multicast message log and retag the results specified in each multicast message. The results specified by a multicast message should be retagged only once, thus each multicast log entry has an associated status regarding retagging. When a multicast message is written to the log its status is retaggable. If the results specified by a multicast message are retagged due to a divergence recovery, the status is altered to not retaggable.

The virtual partition name to be used for retagging is formed from the name of the new virtual partition. If the new virtual partition name consists of level number N and site name S, the retag partition name consists of level number (N-1) and site name S. A retag partition name exists in the gap between the old partition name and the new partition name. All sites that complete the divergence recovery must record the new virtual partition name as part of their partition history; membership in the retag partition is implied by membership in the new virtual partition. Figure V.3 summarizes the processing required to complete a divergence recovery.

```
Receive the commit message for a new virtual partition
If all sites in the new partition report the same partition history
Then
        For each triple <A, N, S> in the commit message Do
                If the sequence number N is > the multicast message sequence number
                                in the local multicast message log for site A
                Then
                        Request a copy of the missing multicast message from site S
                        Log the multicast message
                        Process the multicast message
                EndIf
        EndFor

        For each lock held as a primary site Do
                If the lock was requested by a site that is not a member
                                of the new virtual partition
                Then Release the lock
        EndFor

        For each locally suspended request in the recovering federation Do
                If this request has not multicast its results yet
                Then If this request has requested a lock from a site
                                that is not a member of the new virtual partition
                        Then Backout the processing of this request
                EndIf
        EndIf

        If the local site was the name server site for the previous partition
        Then
                For each name reservation Do
                        If the name was reserved by a site that is not a member
                                        of the new virtual partition
                        Then Release the name reservation
                EndFor
        EndIf

        Commit to membership in the new virtual partition
EndIf
```

Figure V.3 - Algorithm for Committing a Divergence Recovery.

The protocol for divergence recovery as presented is sufficient to maintain mutual consistency among the sites in the newly formed virtual partition.

Proposition 5: DHSS maintains mutual consistency within each virtual partition formed by a divergence recovery.

Proof of Proposition 5: The sites that transition from a partition P_1 to the new virtual partition P_2 by a divergence recovery are mutually consistent if (a) they eventually process the same set of requests and (b) each set of conflicting requests has the same $\rightarrow$ (precedes) relationship on every site in the new virtual partition P_2.

(a) By lemma 4, all sites in the new partition will eventually have processed the same set of requests.

(b) When a site S changes membership from a virtual partition P_1 to a virtual partition P_2, we can divide the requests executed by site S into three sets: the requests executed while S was a member of partition P_1, the requests represented by the multicast messages propagated to site S during the divergence recovery, and the requests executed while S was a member of partition P_2. By proposition 2, we know that the $\rightarrow$ relationship among all conflicting requests executed solely in partition P_1 or solely in partition P_2 is the same at every site that was a member of partition P_1 and partition P_2. The requests not yet accounted for are those whose processing spans the two partitions P_1 and P_2. There are two categories of requests whose execution may span the two partitions: (i) those requests whose associated multicast messages may be propagated during the divergence recovery and (ii) those requests that have not multicast their results and are suspended during divergence recovery.

(i) Requests whose multicast messages are propagated during the divergence recovery are said to execute in some arbitrary order on the remote sites because they are not under the control of any virtual partition. By lemma 3, we know that the set of requests represented by the propagated multicast messages are non-conflicting requests. The order in which propagated multicast requests are executed does not alter the $\rightarrow$ relationship for a set of conflicting requests. Divergence recovery requires that every propagated multicast message must be processed before committing to the new partition P_2. If a request R associated with a propagated multicast message conflicts with a request C, then the request C must have been processed in partition P_1, or the request C will be processed in partition P_2. If request C was processed in partition P_1, then $C \rightarrow R$ at every site belonging to partition P_2. If request C will be processed in partition P_2, then $R \rightarrow C$ at every site belonging to partition P_2.

(ii) All requests known to the system are suspended when divergence recovery begins. The recovery process will selectively backout suspended requests. When the recovery is completed, some of the requests will resume from their point of suspension, those that were backed out will resume by restarting. All requests that were selected for backout will be processed solely in the new partition P_2. By proposition 2, normal request processing maintains mutual consistency among these and all other requests processed in partition P_2. Let $R = \{ r_1, r_2, ..., r_k \}$ be the set of requests that were not selected for backout by the divergence recovery protocol. The set of locks required by request r_i can be divided into two sets. Define H_{r_i} to be the set of locks granted to request r_i during partition P_1 prior to the divergence recovery. Define W_{r_i} to be the set of locks that request r_i wants but has not acquired at the time of the divergence recovery. Consider a request r_i in R and a request c which may or may not be in R. If r_i and c conflict, let X be the set of items on which r_i and c conflict. That is, $X = (H_{r_i} \cup W_{r_i}) \cap (H_c \cup W_c)$. If, at the time of the divergence recovery, $X \cap H_{r_i}$ is non-empty, then c is blocked by r_i in partition P_1. The set of primary sites that have granted the set of locks in H_{r_i} during partition P_1 will honor those locks during partition P_2; therefore, $r_i \rightarrow x$ on every site in P_2. Similarly, if $X \cap H_c$ is non-empty, then r_i is blocked by c in partition P_1. These locks will be honored in partition P_2; therefore, $c \rightarrow r_i$ on every site in partition P_2. If $X \cap H_{r_i}$ is empty and $X \cap H_c$ is empty, the $\rightarrow$ relationship between r_i and c will be determined by the primary sites in partition P_2. By proposition 2, r_i and c will have the same $\rightarrow$ relationship on every site in partition P_2. Therefore, all requests suspended by the divergence recovery will have the same $\rightarrow$ relationship on every site in partition P_2.

All sites in P_2 eventually process the same set of requests and each set of conflicting requests processed in P_1, or P_2, or between P_1 and P_2, have the same $\rightarrow$ relationship on every site in P_2. Therefore, DHSS maintains mutual consistency among the sites in a virtual partition formed by a divergence recovery.

5.2 Merge Recovery

Two or more virtual partitions merge into one virtual partition when a site recovers from a crash, or a communication failure is corrected, or a new site enrolls in a federation. First, the correction of the failure or the enrollment must be detected. Next a new virtual partition is formed, and finally any inconsistencies and conflicts among the sites in the new virtual partition must be corrected.

If a site wishes to enroll in a federation that is currently unknown at that site, the site is easily enrolled by a merge recovery. The merge recovery brings the new site into the federation, propagates all of the directory information to the new site, and merges the site into a virtual partition for that federation. An enrolling site E causes a merge recovery to take place by acting like a crashed site that was in a single site partition named 0E.

If the merge recovery is necessitated by the restart of a crashed site, the failed site must perform a local recovery to stabilize itself prior to merging with other sites in the federation. We will begin our discussion of merge recovery with the protocol for the restart and stabilization of a crashed site. Next we will discuss a modified two phase negotiation for merging virtual partitions. The detection and resolution of conflicts among the merging sites will be discussed in the next chapter.

5.2.1 Crash Site Recovery

When a crashed site is restarted, that site must bring the local DHSS system directory to a stable state prior to communicating with other sites. The purpose of the stabilization process is three fold: first, we must test for media failure due to the site crash; second, we must ensure that each request was executed atomically; third, we must ensure that results committed by this site are already known by other sites or will be propagated to all other sites by future merge recoveries. One side effect of a site crash is that all locks and name reservations recorded at that site will be lost. As part of the stabilization process, the recovering site will form a new virtual partition containing only the local site. Throughout the stabilization process, the recovering site will discard all messages received over the

network and will not initiate any sessions with local clients. Once the site has stabilized as a single site partition, a "finder" will attempt to communicate with other known sites in the federation to force a merge recovery. We will now present in greater detail, the algorithm for crash site recovery.

5.2.1.1 Media Failure in DHSS

Testing for media failure must be the first step in crash site recovery. A stable storage device may experience failure in varying degrees. If any portion of the directory is deemed to be unreadable then we claim that the storage system has experienced a media failure. A severe failure would require that the storage device itself be replaced. In such a case the storage system cannot make an automatic recovery because all information has been removed. Recovery may be achieved by the intervention of a local client executing an enroll request for each federation in which the site was formerly a member. If the failure renders only portions of the storage device unusable the storage system may detect this by attempting to scan the entire DHSS system directory. System restart always begins with a full scan of the local system directory.

The storage system can recover from a partial media failure by building a new system directory based on re-enrollment in each federation in which the site was formerly a member. To carry out this recovery automatically, the storage server must be able to read all of the directory records describing federations and the site and users belonging to those federations. Without this information the storage server will not know which federations to re-enroll in, which sites to contact, and which users to enroll. The readability of this information is not guaranteed but the probability that it can be read is greatly increased by duplicating the federation, site, and user information in the local system directory. With this approach we must designate one copy to be the primary copy for use during non-failure mode processing. Maintenance of the duplicate copies is not a problem, as the extra copies need to be modified only when a user or site secedes from a federation.

If the storage system determines that a partial media failure has occurred, the system will read as many federation, site, and user records as can be located in the system directory. The storage system will then build a new system directory by re-enrolling the local site in

each federation found in the old directory. The users found in the old directory will be re-enrolled in their respective federations. The mechanics of enrollment and re-enrollment are part of a merge recovery and will be discussed in detail later in this chapter.

Traditional approaches to the problem of media failure involve checkpointing with logging (as discussed in chapter two), or making multiple copies of the entire system directory on distinct stable storage devices. In an environment of workstations we do not have the free storage space to duplicate the system directory at each site; we rarely have multiple permanently mounted stable storage devices on a single machine; and we do not want to invest the time to construct and maintain a checkpoint or duplicate local copies. Our approach to media failure is inexpensive in terms of storage space as we duplicate only the federation, site, and user records. The time cost during normal processing is non-existent. A large time cost is incurred only if the recovery must be performed.

Given the overall low cost in time and space, it is not surprising that the DHSS automatic recovery from media failure is not a totally successful recovery. The protocol falls short in two respects.

1. When a site is forced to re-enroll in all of its federations due to a partial or complete media failure, all requests that were being processed at the time of the site crash are lost.

2. When a federation has only one site as a member, there is no possibility of re-enrolling.

The problem of lost requests is partially solved by the use of an undo/redo log. If an undo/redo log is maintained for all currently executing requests and if the log information is readable, after re-enrolling, the system may attempt to redo those requests that were in progress at the time of the crash. The use of undo/redo log information is discussed in more detail in the next section.

5.2.1.2 Atomic Execution of Requests

To ensure atomicity in the event of a site crash the storage system will maintain a write ahead undo/redo log associated with each request currently being processed by the local site. The log for a request will contain the original request message and a report of the new and modified values the request intends to write to the system directory. Log information must be written to stable storage before any of the modifications it reports are written to stable storage.

When all locks, name approvals, and implicit alias numbers have been obtained, the local processing of a request consists of nine steps.

1. Calculate all the new and modified values that will be written to the DHSS system directory.
2. Write the original request message and all intended modifications to an undo/redo log for this request.
3. Commit the undo/redo log information to stable storage by a forced write.
4. Carry out the modifications on the DHSS system directory.
5. Queue for the ability to multicast the results of this request.
6. When the multicast service has been seized, then ensure that the system directory modifications have been committed to stable storage by a forced write.
7. Initiate a multicast of the results of this request.
8. Notify the requesting client of the outcome of the request.
9. Free the undo/redo log space associated with this request for reuse by new requests.

When recovering from a site crash the storage system must undo and redo requests that have not been committed and multicast to other sites. If the processing of a request has not reached the multicasting step (i.e. step 7), then the results of the request should be undone and the request should be re-executed. Once a request has reached the multicast stage its results are known outside of the local site and must not be undone. If the processing of a

request has been committed locally, the results have been multicast to the other sites in the virtual partition, and the requester has received notification of the outcome of the request, then the undo/redo log information associated with this request may be deleted from the log. This means that only a minimal amount of stable storage space must be dedicated to the undo/redo log.

Undo is accomplished by scanning the current undo/redo log entries. If a request must be undone, then the undo information is used to remove each reported modification from the local system directory. Each request whose results are undone should be marked for redos. Requests marked for redo will be processed by the system at a later time.

Once all of the necessary undo operations have been performed, the storage server will perform a consistency test on the local directory. The consistency test is used in an attempt to detect write failures that occurred at the time of the site crash but did not render any records unreadable. The test requires a logical scan of the DHSS system directory. Precisely how to perform the scan and what to check for during the scan are determined by the logical organization and the physical organization used to implement the system directory. By physical organization we mean any linkage between directory records that represents solely physical organization concerns. For example, records may be organized into buckets and the buckets may be chained together. The bucket chains determine a physical organization. By logical organization we mean the logical links between records in the directory. For example, each directory record containing information about a DHSS object must reference another record in the directory specifying that record to contain the information on the principal path for this object. If the record referenced does not contain information about a path then an inconsistency has occurred. Similarly, each record describing a path in a DHSS object will contain a reference to a record that describes the current instance in that path.

The consistency test begins by scanning all physical linkage in the system directory. If the physical linkage is not traversable, then the storage server will claim that an undetected write failure has occurred. If the physical linkage is traversable, then the logical linkage is traversed and the record types are verified. If a logical link is broken or a record type invalid then the storage server will claim that an undetected write failure has occurred. The logical linkage is on a federation by federation basis. If the physical linkage is on a federation by federation basis, recovery from an undetected write failure can be achieved by performing a

media failure recovery for an individual federation. The storage server would cache a list of local users who are members of the ailing federation, free all system directory entries for the federation, and re-enroll the local site and the local users in the ailing federation. If the physical linkage is not on a federation by federation basis then inability to traverse the physical linkage will result in a full media failure recovery.

5.2.1.3 Future Inconsistencies and Establishing Communications

Before attempting to establish communications with other sites the recovering site must ensure that the results recorded in the multicast message log entries are known by the other sites in the federation or will be propagated to the other sites by future merge recoveries. This is accomplished by retagging the results of those requests in the multicast message log with a post-failure virtual partition name. A retag partition name must be constructed for this purpose. The recovering site must know the name of the virtual partition it was a member of at the time of the crash. If this previous partition contained only the crashed site, then no new virtual partition should be formed. The retag partition name is formed by subtracting one from the level number of the previous partition level number. If the previous partition was not a singleton, then the site should construct a new virtual partition name based on the previous partition name. The retag partition name is formed by subtracting one from the level number of the new partition level number. The retag partition name should be used to retag the results specified by each multicast log entry that has not been retagged previously.

Once all necessary undo operations have been performed, all necessary multicast results have been retagged, and the site has committed to a singleton virtual partition, a finder process is started. The finder will cycle through the federations known to the local site and attempt to communicate with sites belonging to those federations. For each federation the finder will send a liveness message to one of the member sites, set a timeout for receiving a virtual partition invitation for that federation, and wait. If the invitation arrives within the specified time limit, a merge recovery will be executed. If the time limit expires before an invitation is received, the finder will attempt to communicate with a different site belonging to the federation. If all sites in a federation fail to respond, then the

recovering site will begin processing as a singleton virtual partition. The first step in processing is to redo all those requests that were marked for redo during the undo phase of crash recovery. Once those requests have been redone, the system will become available for new requests. The finder will periodically send a liveness message to the non-communicative sites in the federation. Figure V.4 summarizes the complete algorithm for crash site recovery.

Algorithm for Crash Site Recovery

```
Perform a physical readability test on the system directory for media failure
If the directory is not found
Then Create an empty directory and wait for client requests
If some portion of the directory is not readable
Then
        Read and cache all readable federation, site, and user records from the directory
        For each unique federation Do
                Execute an enroll request for each user in this federation
        EndFor
EndIf
If all parts of the directory are readable
        For each request recorded in the undo/redo log Do
                If the request has not been committed and multicast
                Then
                        Removes all results of the request from the local system directory
                        Mark this request log entry for redo
                EndIf
        EndFor
        If the previous virtual partition was not a singleton partition
        Then Construct a new virtual partition name (previous level# + 2, local site)
        Construct a retag partition name (current partition level# -1, local site)
        For each multicast message log entry Do
                If the multicast message results have not been previously retagged
                Then
                        Retag these multicast results with the retag partition name
                        Mark the multicast message log entry as retagged
                EndIf
        EndFor
        Commit to a single site virtual partition
        Activate the finder to locate other sites using liveness messages
        For each entry in the undo/redo log Do
                If the enry is marked for redo
                Then
                        Redo the request
                        Unmark the log entry
                EndIf
        EndFor
EndIf
```

Figure V.4 - Algorithm for Crash Site Recovery.

5.2.2 Detection of Failure Correction

Two or more virtual partitions may merge only after the recovery of a crashed site or the correction of a communication failure has been discovered by some site. The sending and receiving of liveness messages will be the general mechanism used for detecting the correction of such failures. Continually sending liveness messages among all sites would cause a proliferation of messages on the network. We propose to send a liveness message only to those sites that we believe we are unable to communicate with. The service request messages and the multicast messages are really a form of liveness message sent exclusively to those sites we believe we can communicate with. The receiving of a liveness message serves as a notification that a failure situation has been corrected. If a site receives a liveness message from another site, the receiving site must attempt to initiate a new virtual partition.

Liveness messages are sent and received by a finder process of the storage system. The purpose of the finder is to communicate with sites not in the current virtual partition of a federation. Figure V.5 summarizes the algorithm used by the finder. The finder executes periodically. If a federation is in normal processing mode, the finder will try to communicate with sites outside of the current partition. When a federation is performing partition negotiation and recovery, the finder is dormant for that federation. At each execution, the finder will cycle through the federations known by the local site. For each federation we will determine the set of excluded sites. The excluded sites are those sites that are members of the federation and are not members of the current virtual partition for that federation. A liveness message will be sent to one of the excluded sites. Upon the next execution of the finder, if the set of excluded sites for a particular federation is the same, we will determine the next site in some ordering of the set and select it as the destination for a liveness message. It is sufficient to send only one liveness message per federation per execution of the finder because if a merge recovery is initiated the invitation to join a new virtual partition is always sent to all known sites; thus, all the sites that are able to merge will be located by using only one liveness message.

```
Algorithm for Detecting Failure Correction

For each federation known by the local site Do
    If this federation is currently in recovery
    Then Skip this federation
    Else
        New excluded sites = { sites that are members of this federation
                and are not member of this federation's current virtual partition }
        Order the new excluded sites by name
        If the set of new excluded sites = the previous set of excluded sites
        Then
            Determine which site was the destination of the last liveness message
            Send a liveness message to the next site in the ordered set
            Mark the destination site as the last destination of a liveness message
        Else
            Send a liveness message to the first site in the ordered set
            Mark the first site as the last destination of a liveness message
        EndIf
    EndIf
EndFor
```

Figure V.5 - Finder Algorithm for Detecting Site and Communication Recoveries.

5.2.3 Merging Virtual Partitions

When two or more virtual partitions merge, each virtual partition may have processed requests that the other partitions have not processed. The requests processed by disparate virtual partitions will be classified as conflicting requests or missing requests.

Definition: Two requests R_1 and R_2 conflict if they are executed in separate virtual partitions and the intersection of the set of objects and paths altered by R_1 with the set of objects and paths altered by R_2 is non-empty.

For example, two update requests conflict if they update the same instance of user data; two assign requests conflict if they assign the principal path of one DHSS object to be different paths in that object. All non-conflicting requests are missing requests.

The goal of merge recovery is to achieve mutual consistency among the merging sites. This requires the resolution of conflicts presented by conflicting requests and the propagation of the results of all missing requests. To accomplish this, information from the system directory must be exchanged between each pair of unique virtual partitions. This means that there must be one system directory that represents the view held by all sites

The Virtual Partition Initiator sends a Commit message containing:
- Name of the new virtual partition
- For each unique virtual partition history reported, a list containing:
  - The virtual partition history
  - A list of the sites that reported this history
  - The representative site for this partition history
  - For each site reported by a site with this history, one triple containing:
    - site name
    - multicast message sequence number
    - name of a holding site that reported this sequence number

Figure V.6 - Contents of a Virtual Partition Commit Message.

within a single virtual partition. The partition initiator selects one site from each of the merging virtual partitions to be the representative system directory for that partition. The list of representative sites will be included in the commit message sent by the partition initiator. Figure V.6 states the complete contents of the commit message sent by the virtual partition initiator.

After the two phase negotiation of a new virtual partition, the third phase of merge recovery will exchange information among the merging sites. Each representative site provides information that represents a mutually consistent view held by all of the sites in the representative's partition. Each of the merging virtual partitions can attain mutual consistency with respect to requests that have been multicast by performing the third phase of a divergence recovery among the sites within the partition.

Consistency with respect to those requests that have been multicast is not sufficient. The divergence recovery brings each partition to mutual consistency on the level of multicast messages. Each site may have partially processed requests which the other sites know nothing about. This presents a problem for a merge recovery. In a merge recovery, the information provided by a representative site is in the form of copies of records from the representative's system directory. This information is not at the same level of abstraction as multicast messages. All of the merging sites must process the exchanged information at this lowest level of abstraction. To process this information correctly, each site must be internally consistent. That is, no system directory may contain the partial results of requests that have not been multicast yet. During normal request processing we used mechanisms to achieve the atomic execution of one request with respect to all other requests. A merge

recovery is not a request; therefore it circumvents these atomicity mechanisms by looking directly at the suspended state of a system directory. This suspended state must be an internally consistent state. Internal consistency can be achieved by undoing all suspended requests that have not been multicast. Each undone request should be marked for redo.

A merge recovery may force a set of requests to be undone and redone at a later time. These requests may have obtained locks or name reservations prior to the time merge recovery was initiated. These locks must be released by the merge recovery process otherwise the request will become hung when it is redone. To avoid hung requests, every site participating in the merge recovery must release, for the federation undergoing recovery, all locally held locks and name reservations.

Once each merging virtual partition has achieved mutual consistency and each merging site has achieved internal consistency, the representative sites will scan their local system directory to locate all information that must be exchanged among the merging sites. Every site will send a message to each virtual partition representative requesting the information being provided by each virtual partition. We will refer to this information as the change list; it is indeed a list of all the changes that are known in a source partition and are unknown in one or more of the merging partitions. The contents of and the processing of change lists will be discussed in detail in the next chapter. Once all change list information has been processed, each site will commit to the new virtual partition. Once committed to the new partition, the requests marked for redo in the undo/redo log are re-initiated, and new requests are accepted and processed. Figure V.7 summarizes the algorithm for processing the third phase of a merge recovery.

Algorithm for Processing a Partition Commit Message

```
Receive the commit message for a new virtual partition
If multiple partition histories are reported
Then
        Perform propagation of multicast messages as in divergence recovery
        For each entry in the undo/redo log Do
                Remove all results of the uncommitted requests from
                                the local system directory
                Mark the entry for redo
        EndFor
        Clear all locks held by the local site for this federation
        Clear all name reservations held by the local site for this federation
        If the local site is designated as a representative
        Then
                Calculate the tag set for the partition being represented
                Scan the local system directory and construct a change list
        EndIf
        For each virtual partition history in this merge Do
                Obtain the change list from the representative for this partition history
        EndFor
        Perform resolution on the set of change lists and incorporate the results
                                into the local system directory
        Commit to membership in the new virtual partition
        For each entry in the undo/redo log Do
                If the entry is marked for redo
                Then
                        Redo this request
                        Unmark this undo/redo log entry
                EndIf
        EndFor
EndIf
```

Figure V.7 - Algorithm for Merging Virtual Partitions.

CHAPTER SIX
CONFLICT DETECTION AND RESOLUTION

To achieve mutual consistency among a set of merging partitions, we must detect, propagate, and incorporate missing results and we must detect, resolve, and incorporate conflicting results. We will now discuss in detail the algorithm employed by merge recovery to achieve a mutually consistent view of the data stored by the system.

6.1 Merging Partition Histories

In a merge recovery, the representative site for each partition must construct a list of information that will be useful in achieving mutual consistency among the merged sites. The representative site cannot provide a list of all operations executed within the partition represented because the storage system does not maintain an audit trail of system operation. However, an audit trail of system operation is not required and may even be undesirable. When multiple partitions are merging, it is not essential to have knowledge of every operation executed in a virtual partition. What is essential is knowledge of which items have changed value one or more times within a partition, and what the current value of each such item is. Consider the following example. In one partition a client performs an assign operation on an object O. The assign request alters the name mapping of object O to path p_1. A short time later in that same partition, object O is assigned to path p_2. When a merge is performed, knowledge that object O was assigned to p_1 is of little or no use. The important information is that the mapping of object O to its principal path was altered and O currently maps to p_2. Using the knowledge that O was assigned to p_1 and then p_2 would require that each merging site process both of these actions, even though the results of the first are no longer visible. Further, the actions must be performed in order so that all merging sites conclude that object O currently maps to path p_2. Therefore, a representative site need only determine which items have changed value unbeknown to other partitions, and what the newest value is.

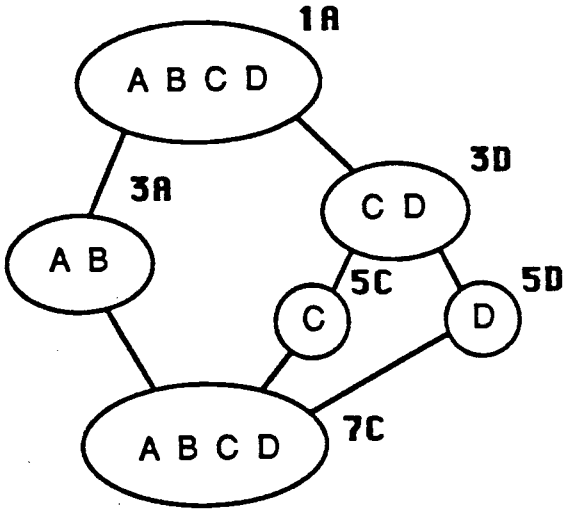
Determining the current value of an item is very simple. The current value is stored in the system directory of the representative site and is consistent with the other sites in the represented partition due to the execution of a divergence recovery. Determining which items have changed value unbeknown to the other partitions requires the maintenance of additional information. We associate a tag with each changeable item. The tag value is the name of a virtual partition. When an executing request changes the value of an item, the value of the partition tag associated with that item will also be changed to the name of the current virtual partition. This tag value will be used by the storage system to determine which items have changed without the knowledge of one of the merging partitions.

Federation Sites = { A, B, C, D }

Partition 3A = { A, B }
Partition 5C = { C }
Partition 5D = { D }

VPH(3A) = { 1A, 2A, 3A }
VPH(5C) = { 1A, 2D, 3D, 4C, 5C }
VPH(5D) = { 1A, 2D, 3D, 4D, 5D }

Representative(3A) = B
Representative(5C) = C
Representative(5D) = D



VPH(7C) = { 1A, 2A, 2D, 3A, 3D, 4C, 4D, 5C, 5D }

Figure VI.1 - Example of Partitioning Behavior in a Four Site Federation.

Figure VI.1 shows a federation of four sites A, B, C, and D, the partitioning behavior of those sites, and the virtual partition history (VPH) of the three virtual partitions 3A, 5C, and 5D. We wish to investigate a merger of the three partitions. By examining the virtual partition histories, we observe that all of the partitions claim membership in virtual partition 1A. From this we conclude that every item that has not been changed since partition 1A has a value which is known by all of the merging sites. Sites A and B claim membership in virtual partitions 2A and 3A; sites C and D do not claim membership in partitions 2A and

3A. From this we conclude that every item that was changed by the sites in partitions 2A and 3A has a value which is unknown by sites C and D. Similarly, every item that was changed by the sites in partitions 2D and 3D has a value which is unknown by sites A and B. Site C is the only site that claims membership in partitions 4C and 5C. Thus, each value changed by the site C during the existence of partitions 4C and 5C has a value that is unknown by sites A, B, and D. Similarly, each item changed by site D during the existence of partitions 4D and 5D has a value that is unknown by sites A, B, and C.

For the merge recovery which forms partition 7C we select site B as the representative for partition 3A, site C as the representative for partition 5C, and site D as the representative for partition 5D. Site B will provide information on all items having an associated partition tag value of 2A or 3A. Site C will provide information on all items having an associated partition tag value of 4C or 5C. Site D will provide information on all items having an associated partition tag value of 2D, 3D, 4D, or 5D. Even though the sites in partition 5C and the sites in partition 5D were members of 2D and 3D, the changes made during partitions 2D and 3D must be included in the merge information because these changes are not known by the sites in partition 3A and may conflict with changes performed during partitions 2A and 3A. Note that the changes performed during partitions 4C, 5C, 4D, and 5D cannot conflict with the changes performed during partitions 2D and 3D because these changes were performed in full knowledge of the changes performed during partitions 2D and 3D.

Figure VI.2 states the algorithm by which a partition representative calculates the set of virtual partition tag values which mark changed items that are of interest for the merge. We note that the representative of partition 5C and the representative of partition 5D are both capable of supplying merge information tagged with the partition name 2D or 3D. It is desirable that only one of these two representatives supply this information. This will reduce the amount of information sent over the network to all sites; but more importantly it reduces the amount of redundant information that must be processed by each of the merging sites. The algorithm in figure VI.2 assigns to the representative of the virtual partition with the largest name the task of supplying all information that is known by that partition and is unknown by at least one of the merging partitions. All other representatives must supply just the information that is known solely by the sites in their partition.

Let N be the number of partitions in this merge
Order the merging partitions by their current virtual partition name
Let J be the position of the local site's partition in the ordering
If your current virtual partition name < the largest virtual partition name among the merging partitions
Then

Set of virtual partition tags =

$$\left\{ \bigcap_{i=(1,J+1)}^{(J-1,N)} \left[VPH\left[local\ site's\ partition \right] - VPH\left[i\text{-}th\ merging\ partition \right] \right] \right\}$$

Else

Set of virtual partition tags =

$$\left\{ \bigcup_{i=1}^{N} \left[VPH\left[local\ site's\ partition \right] - VPH\left[i\text{-}th\ merging\ partition \right] \right] \right\}$$

Figure VI.2 - Algorithm used by Representative Sites to Calculate a Tag Set.

A representative site begins by calculating its virtual partition tag set for the merge. Next the representative constructs a change list by scanning the local system directory selecting a copy of each record that contains a tag value in the representative's tag set. Every merging site will request the change list information from every representative. Missing results will be added to each local directory. Conflicting results will be resolved and added to each local directory. When a site has completed the merge recovery and commits to membership in the new virtual partition, that site will have as its partition history the union of all of the partition histories in the merge.

6.2 Change Lists

The information provided by the representative sites must be sufficient to detect missing results, to detect conflicting results, to unilaterally resolve conflicting results, and to ensure the achievement of mutual consistency in future mergers. The information to accomplish these goals is of three types: the current value of a modified item, the partition tag associated with the modified item, and control or context information.

Most of the information contained in a change list is selected for inclusion based on the value of an associated partition tag. Previously we discussed a partition tag as an attribute associated with a modifiable item. We will now specify what the modifiable items are in the DHSS environment. The items undergoing change in our versioned storage system are the system directory entries rather than the user data. If a client modifies and updates user data, a new version will be created; the storage system will store the new version of the user data and add new records to the system directory. If a client changes the status, accessibility, or ownership of user data, then the storage system will modify an existing record in the system directory. Sites maintain mutual consistency by maintaining the mutual consistency of their system directories. Thus, in the DHSS environment we must associate partition tags with fields in a system directory record. A field must have an associated partition tag if the field may be modified as the direct result of a client request and the field stores a value that must be identical on every site. An example of a field that requires an associated partition tag is the path record field which references the current instance for the path. This field is modified when a client updates the path, and each site in the current virtual partition must have the same value for this field. An example of a field that does not require an associated partition tag is the field which specifies the local representation (full copy, differenced copy, or no copy) of an instance of user data. Because the field contains strictly local information, the value will vary from site to site. We assume that if a system directory record contains one or more partition tags whose value is in the tag set, then the entire directory record will be included in the change list. This assumption is not necessary but it is sufficient.

The change list is constructed by scanning the entire system directory. As the directory is scanned two distinct portions of the change list are constructed: control information about group updates executed during the partitions of interest, and copies of all system directory records modified during the partitions of interest. The control information about group updates gives the group update token, the identity of the paths or objects updated, the identity of the instances created by the update, the identity of the predecessor instances, and which group update rule was used to process the update. How this information is used to resolve group update conflicts is discussed later in this chapter. The second part of the change list must be clustered and ordered by objects. For each path record in the change list all instance records pertaining to that path will be grouped with the path record. All of the paths within an object will be clustered together and ordered by the immutable token associated with a path. If an object record appears in the change list it will

be grouped with the path records for that object. Finally, object groups are sorted by the immutable token associated with each object. Ordering the change lists in this manner allows each site to perform N-way merge processing on N change lists. If a record appears in only one of the incoming change lists, then only one partition modified that record, no conflicts exist and the missing results should be added to the local system directory. If a record appears in multiple change lists, a conflict may exist.

The control information on group updates is necessary because it clusters the group update information by groups of instances. The second part of the change list clusters all information by the object it is associated with. Providing the initial control information avoids the need to repeat all of the instances updated by a group update with each instance as it appears clustered with its object.

General Algorithm for Processing Change Lists

```
For each change list Do
    While this change list contains group update information Do
        Read and save the information
    EndWhile
EndFor
Determine the smallest object token not yet processed by the merge, call it O
Consider the object set for O from each change list that reports on object O
Create an empty model for the object O
For each non-conflicting change Do
    Add the results of the change to the model records for object O
    Copy all associated partition tag values into the model records
EndFor
For each set of conflicting changes Do
    Resolve the conflict
    Add the results of the conflict resolution to the model records for the object O
    Set all associated partition tag values in the model records to the
                        name of the newly formed virtual partition
EndFor
For each record in the model Do
    Incorporate the model for object O into the local system directory
EndFor
```

Figure VI.3 - General Algorithm for Processing Change Lists during a Merger.

Figure VI.3 shows a high-level algorithm for processing the change lists during a merge. All control information about group updates is read and saved for use in resolving group update conflicts. Next the directory record images are processed. For each object O appearing in any change list, the merge and resolution algorithm builds a model object to be incorporated into the local system directory. The initial model consists of an empty object

record. All changes prescribed by the change lists are performed on the model object record. If the change is non-conflicting, the new results are copied into the model record and the partition tag field associated with the change is copied into the model record. If the changes are conflicting, the conflicts are resolved, the results are copied into the model record(s) and all partition tag fields associated with values that are determined by the resolution algorithm are set to the name of the new virtual partition being formed. Thus, those sites that are merging to form the new virtual partition are the sites that have knowledge of the results produced by the resolution algorithm. The resolution algorithm may require the addition of new alternate versions to the object. When alternate versions are added to the object O, model path records are added to the model and associated with the model tree record. Model instance records are added to the model in a similar manner. When all change list information about object O has been processed, all model records are incorporated into the local system directory, the model is reinitialized to empty, and the next object specified by one or more change lists is processed.

> **Lemma 6:** For all non-conflicting results in the change lists, the merge algorithm produces mutually consistent results in the model records on all merging sites.

Proof of Lemma 6: All sites participating in the merge receive the same change lists. A result is non-conflicting if it appears in only one change list, or is the identical result in multiple change lists. Given that all sites receive the same change lists, a non-conflicting result R is detected by every merging site. The merge algorithm copies the non-conflicting results as is into the model records; therefore, the model records at all merging sites are mutually consistent with respect to non-conflicting results.

The algorithm for incorporating the model into the local directory is presented in figure VI.4. Each record in the model is processed individually. If the record specified by a model record exists in the local system directory, then the local copy is modified by copying, from the model record to the local copy, each field or set of fields that were modified by any of the merging partitions. A modified field is identified by an associated partition tag field with a value in the range of interest for the merge. The partition tag values in the range of interest for a merge are the union of the partition tag sets of all of the representatives in the merge and the name of the newly formed virtual partition. If the record specified by the model record does not exist in the local system directory, then all fields in the model record that pertain to local information (e.g. the physical representation of a local copy of a data

instance) must be set to default values and a copy of the model record is added to the local system directory.

```
Algorithm for incorporating a model into the system directory

    For each record in the model Do
        If the corresponding record exists in the local system directory
        Then
            Read the local record
            For each partition tag field in the model record Do
                If the partition tag field value is
                                        in the set of tags of interest
                Then Copy from the model record into
                                    the local record the field(s) associated
                                    with this partition tag field and the
                                    partition tag field
            EndFor
            Write the modified local record to the local directory
        Else
            If the record contains fields of local significance only
            Then Fill these fields in the model record with default values
            Add the model record, as is, to the local directory
        EndIf
    EndFor
```

Figure VI.4 - Algorithm for Incorporating a Model into the Local Directory.

Lemma 7: If a model record R produced by the resolution algorithm is mutually consistent on every merging site, then the incorporation algorithm will produce a mutually consistent record corresponding to R in the local system directory of every merging site.

Proof of Lemma 7: Let P_1, P_2, ... P_n be the merging partitions. Let R be a model record that is mutually consistent at every merging site. Let f be a field in the record R. With respect to the resolution algorithm, the field f is of one of two types: f is a field for which the resolution algorithm did not specify a value; or f is a field whose value was determined by the resolution algorithm (i.e. the field value was a non-conflicting result or was produced as the resolution of two conflicting results).

(a) Let f be a field whose value was not specified by the resolution algorithm. Let pt_f be the partition tag field associated with the field f. Given that the value of f is not specified by the resolution algorithm, the value of pt_f is not in the range of interest for the merge. Let S_1 and S_2 be two of the merging sites. Assume after performing the incorporation algorithm on R, the field f has the value v_1 on site S_1 and the value v_2 on site S_2. The resolution algorithm did not provide a value for f so v_1 and v_2 must

have been the value of f at sites S_1 and S_2 respectively prior to the merge recovery. If S_1 and S_2 were in the same virtual partition the last time f was modified then S_1 and S_2 are mutually inconsistent with respect to f. This is a contradiction to corollary 3 that states that normal processing maintains mutual consistency among all sites in a virtual partition. If S_1 and S_2 were members of distinct partitions when S_1 modified f to the value v_1 or S_2 modified f to the value v_2 then pt_f was also modified by S_1 or S_2 to be the name of the current virtual partition. Assume it was S_2 that altered the value of f to be v_2. The value of pt_f on site S_2 must be a virtual partition name that is in the range of interest for the merge. Therefore, the value v_2 would be considered by the merge and the resolution algorithm would determine a value for f. Contradiction.

(b) Let f be a field whose value is determined by the resolution algorithm. The value of pt_f must be in the range of interest for the merge. By the definition of the incorporation algorithm, the value of f is copied as is into the local system directory record for R. Given that F is mutually consistent in the model record for R, then f is mutually consistent in the local system directory record for R at every merging site.

Therefore, the incorporation algorithm processes mutually consistent fields from model records to produce mutually consistent fields in the local system directory of every merging site.

6.3 A Simple Merge Example

To obtain a better intuitive feeling of what information is placed in a change list and what results are produced from processing the change lists, we will begin our discussion with a simple example.

Assume an initial environment consisting of a federation of two sites A and B executing in a virtual partition called 1A. The default replication factor for the federation is two. A client on site A creates an object O. Sites A and B add three new records to their local system directory. One record describes the object O; a second record describes the principal path in O; a third record describes the current instance in the principal path of O. The object record must contain some reference to the principal path record. Each reference must be implemented as a field in the referencing record. Thus, the object record contains a

field that references the principal path record, and the principal path record contains a field that references the current instance record. We associate a partition tag with these two fields and with the instance record. When the records are created the partition tags are set to the current virtual partition name 1A. Figure VI.5(A) shows the three identical directory records on sites A and B.

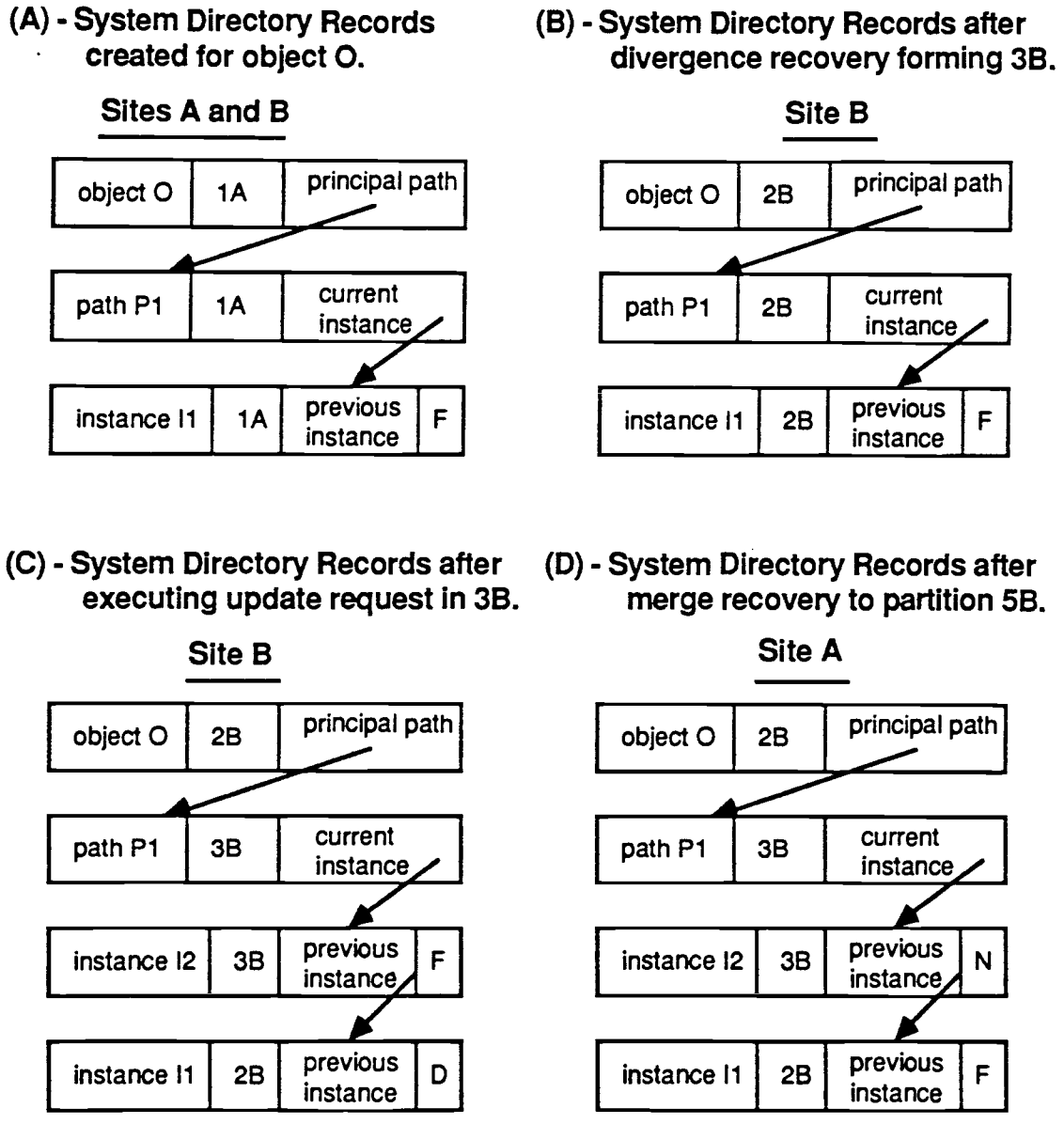


Figure VI.5 - Propagation of a Missing Update in a Merger.

Suppose sites A and B experience a communication failure. Because a copy of object O resides at both sites, the optimistic access rule allows clients at both sites to checkout and update the object. A client at site B checkouts the object O and updates it. When executing the update request site B must obtain a lock from site A, the primary site for object O. Site B detects the communication failure and performs a divergence recovery to form partition 3B. The divergence recovery causes the last multicast message from each site to be retagged with the partition stamp 2B. Site B sent no multicast messages. Site A sent the create multicast message; the results of the create are retagged on site B. Figure VI.5(B) shows the directory records on site B after the divergence recovery. The directory records on site A are not modified. To complete the update request site B elects itself as the pseudo primary site for object O, a new instance record is added to the local system directory, and the instance reference in the principal path record for object O is modified. The partition tag associated with the new instance record is set to 3B. Also, the partition tag associated with the instance reference in the principal path record is set to 3B. Figure VI.5(C) shows the directory records on site B after the execution of the update request.

Assume that the communication failure is corrected. The finder on site B must detect the correction by sending a liveness message to site A. Site A constructs a new virtual partition name 3A and proposes this name to every known site (sites A and B). Site B rejects the virtual partition name 3A and counter-proposes the virtual partition name 5B. Sites A and B accept the invitation to form partition 5B. Site A reports a partition history of 1A, and site B reports a partition history of 1A, 2B, and 3B. Site A is the representative for partition 1A. The tag set for site A is empty, thus site A does not scan the local system directory. Site B is the representative for partition 3B. The tag set for site B is { 2B, 3B }. Site B scans the local system directory for records containing partition tag values of 2B or 3B. All four records in the directory on site B contain partition tags of 2B or 3B. The change list constructed by site B begins with update information about instance I2 as the group update control information. Following the control information is a copy of the four directory records on sites B.

Sites A and B each obtain a copy of the change list from site B and the null change list from site A. Because only one change list contains information, there are no conflicts to resolve. Site B has nothing to do because the change list from site A is empty. The system directory records on site B remain the same. Site A incorporates the information from site B's change list into the local system directory. Figure VI.5(D) shows the directory records

on site A after the incorporation. Site A must determine that the object record, the principal path record, and the I1 instance record have been modified by the sites in another partition and should be modified locally. Incorporating the modifications to these three records does not mean that the old versions of the records are simply overwritten. In the case of the instance record for I1, the local representation value must remain as full. The record for instance I2 does not exist locally, thus it is a new record to be added to the system directory. Before adding the record, the local representation field must be set to a default value of none.

6.4 The Philosophy of Conflict Resolution

The resolution algorithm used during merge recovery should meet two requirements.

1. The resolution algorithm must ensure that the mapping from external names to internal data instances is a one-to-one mapping.

2. The resolution algorithm must produce a mutually consistent view of the system among the merging sites.

There are four properties of the resolution algorithm that are desirable from an implementational point of view.

1. The resolution algorithm should be as simple as possible.

2. The resolution should be achieved with a minimum amount of "undo/redo" effort by the system.

3. The resolution should not appear to be ludicrous to any of the clients.

4. The resolution algorithm should not require additional voting by the merging sites; that is, the resolution algorithm should depend solely on the changes that were made, what site initiated the change, and what virtual partition that site was a member of when the change was made. This would allow the resolution algorithm to be employed unilaterally by each site in the merger.

It is not possible to define a resolution algorithm that meets the requirements and possesses all of the desirable properties. Keeping clients contented under all circumstances is impossible. Attempts to minimize the amount of change caused by a merger could require very complex algorithms. Requiring that the resolution algorithm be a non-voting protocol should reduce the complexity of the resolution algorithm, but may decrease the acceptability of the resolution itself.

When two or more virtual partitions merge, two types of request conflicts can occur.

1. Merge-able conflicts are conflicts where the results produced by each conflicting request are modified and added to the system in a non-conflicting way.

2. Direct conflicts are conflicts where the results produced by exactly one of the conflicting requests will prevail and be added to the system.

Figure VI.6 classifies pairs of conflicting DHSS write requests as merge-able conflicts, direct conflicts, or a combination of both.

> **Definition:** Two write request types A and B form a conflict pair (A, B) if there exists a write request A′ where A′ operates on a data object X and A′ is a request of type A, and if there exists a write request B′ where B′ operates on X and B′ is a request of type B, and A′ and B′ are conflicting requests.

The most intricate resolutions involve merge-able conflicts. The merge-able conflicts are resolved by selecting one of the conflicting requests as a winner, modifying the results of the losing conflicting requests, and adding all of the results to the system directory. In the case of direct conflicts, only one result will be selected and added to the system directory. When direct conflicts occur, there is in general no semantic basis for selecting one result to prevail over another result. The goal of the resolution of direct conflicts is to make the best choice (when best can be ascertained) and to make a mutually consistent choice.

The resolution algorithm must resolve all of the conflict pairs classified in figure VI.6. The resolution of each conflict pair is not totally independent of each other. Some conflict pairs are resolved simultaneously (such as (name, name) and (create, create)), because the pairs represent the same kind of conflict and are resolved by a single algorithm. Some conflict pairs must be resolved before others (such as (derive, derive) before (update, update)), because their resolution produces information that affects the outcome of other resolutions.

	Request Pairs
Non-Conflicting	checkout, read, return, name_match
Merge-able Conflicts	(erase_all, X ; where X = name, create, derive, set_permission, grant, deny, assign, new_owner, usurp) (delete, X ; where X = name, create, derive, set_permission, grant, deny, assign, new_owner, usurp) (name, name) (create, create) (derive, derive) (name, create) (name, derive) (create, derive) (grant, grant) (deny, deny) (grant, deny) (set_permission, grant) (set_permission, deny) (update, update)
Direct Conflicts	(assign, assign) (erase_all, update) (delete, update) (new_owner, new_owner) (usurp, usurp) (new_owner, usurp)
Merge-able or Direct Conflicts	(set_permission, set_permission)

Figure VI.6 - Classification of Pairs of Conflicting DHSS Requests.

6.5 The Resolution Protocols

The resolution of the requests processed by N distinct virtual partitions is an N-way merge of the N change lists representing those partitions. Some conflicts involve multiple objects such as group updates or naming conflicts. Each change list begins with control information about group updates. This information is read and saved for use in resolving conflicting group updates. The remaining information in the change lists is processed one object at a time. We begin our presentation of the resolution algorithms with naming conflicts. Next we deal with conflicts that may affect the object record itself, followed by conflicts affecting the paths records, and concluding with conflicts affecting the instance records.

6.5.1 Name Conflicts

The name request, create request, and derive request may be in conflict with each other by adding duplicate user-defined object base names to the system. A naming conflict is a merge-able conflict. As discussed in chapter three, each user-defined name has an invisible suffix consisting of the defining user's name and the name of the site where the creating request was issued. These duplicate user-defined names are merged by making the invisible suffix visible. The addition of the suffix will make the name unique among all names in a federation. Therefore, the resolution of (name, name), (create, create), (name, create), (name, derive) and (create, derive) conflicts is achieved automatically by propagating the results of the requests to each of the merging partitions.

> **Lemma 8:** For the set of names appearing in any of the change lists, the resolution algorithm for resolving name conflicts produces mutually consistent name fields among the model records on every merging site.

Proof of Lemma 8: Each object or path name created by a name request, a create request, or a derive request consists of a user-defined portion, the name of the user making the request, and the name of the site at which the request was submitted. The concatenation of these three parts always forms a unique name because a single site can test for a duplicate name defined by a single user by examining the local system directory records. If the name composed by concatenating the user-defined portion of the name and the user's name is already known on the site, the duplicate name definition is not allowed. Therefore, all three-part names are non-conflicting names. By lemma 6, the resolution algorithm merges all non-conflicting results to form mutually consistent fields in the model records of all merging sites. Therefore, the algorithm for resolving name conflicts by considering all names to consist of three parts produces mutually consistent name fields in the model records of all merging sites.

6.5.2 Assign Conflicts

When building the model object record for an object O, all conflicting assign requests on object O must be resolved. When an assign request is executed, the request modifies an object record in two ways: first, the principal path field is altered, second, a partition tag field associated with the changes to the principal path field is set to the name of the current virtual partition. If multiple change lists contain the object record with a principal path partition tag field whose value is in the range of interest, then an assign conflict has occurred. All assign conflicts are direct conflicts. The merged result will represent the selection of one of the assigns as the winning assignment. The winning assign is the principal path from the change list record whose partition tag value is the largest. Figure VI.7 summarizes the algorithm for resolving conflicting assign requests.

Resolving Assign Conflicts

Select the assign result with the largest associated partition tag
Set the principal path in the model to the selected assigned path
Set the associated partition tag field to the name
of the newly formed virtual partition

Figure VI.7 - Algorithm for Resolving Conflicting Assign Requests.

The resolution algorithm for conflicting assign requests selects the same winning assign requests on every merging site.

> **Lemma 9:** For the set of assign request results appearing in any of the change lists, the resolution algorithm for resolving conflicting assign requests produces a mutually consistent principal path in the model object record on every merging site.

Proof of Lemma 9: By definition of the resolution algorithm, every merging site selects the assign result with the largest associated partition tag value. A partition tag is the name of a virtual partition. The set of virtual partition names is totally ordered; therefore, given a set of partition tag values, there exists a uniquely determined maximum value. Let O be an object. Let PT be the maximal partition tag value associated with the principal path of object O proposed by any change list. Assume that more than one change list proposes the maximal value PT in association with the

principal path in object O. Let $C_1, C_2, ..., C_n$ be the change lists that report PT as the partition tag value in association with the principal path in object O. If $C_1, C_2, ..., C_n$ are also reporting the same principal path for the object O, then all of these change lists are reporting the same assign request. Selecting any one of the these as the winner selects the same result; therefore, all of the merging sites select the same winning assign request. Suppose that C_i reports a principal path different from the path reported in change lists $C_1, ..., C_{i-1}, C_{i+1}, ..., C_n$. The partition tag value PT means that all sites in partition PT had knowledge of the assign request. The change lists $C_1, ..., C_{i-1}, C_{i+1}, ..., C_n$ are reporting a different assign result in partition PT. This may occur only if mutual consistency is lost in virtual partition PT. By corollary 3, mutual consistency is maintained in a virtual partition. Contradiction. All merging sites select the same winning assign result; therefore, the principal path specified in the model object record is the same at every merging site.

6.5.3 Ownership Conflicts

New_owner requests may represent direct conflicts that affect the object record and path records. When a new_owner request is executed two fields are modified in the object record or in a path record: the owner field is changed to a different user, and a partition tag field associated with changing the owner field is set to the name of the current virtual partition. If multiple change lists contain the object record for object O and the partition tag field associated with change in the owner field is in the range of interest, then a new_owner conflict has occurred. The conflict is resolved by selecting the new owner that has the largest value for its associated partition tag field.

New_owner conflicts for paths are managed analogously to new_owner conflicts for objects; however, new_owner requests on paths may conflict with usurp requests on paths. Each path record contains a partition tag field associated with a change of ownership by a usurp request. Only the object owner may perform a usurp request. The object owner is considered to be more powerful than the path owner; therefore, in general, the usurp will prevail over the new_owner request. A difficulty arises when there has been a new_owner conflict for the object itself. In such a case, the winning usurps should be consistent with the

```
Resolving Ownership Conflicts

    If there are new owner conflicts on the object
    Then Add to the model object record the new owner with the largest
                associated partition tag value and the new virtual partition name
    EndIf
    For each object path that appears in any change list Do
        Case of conflicts
        New Owner and Usurp Conflicts:
            If there was a new owner conflict for the object
            Then Add to the model path record the new path owner resulting from
                                the new_owner or usurp request performed in the
                                same partition as the new owner for the object
                                and the new virtual partition name
            Else Add to the model path record the new owner with the largest
                                associated partition tag value and the new
                                virtual partition name
            EndIf
        Usurp Conflicts:
            If there was a new owner conflict for the object
            Then Add to the model path record the new path owner resulting from
                                the usurp performed in the same partition as the
                                new owner for the object and the new virtual partition name
            Else Add to the model path record the new path owner resulting from
                                the usurp with the largest associated partition
                                tag value and the new virtual partition name
            EndIf
        New Owner Conflicts:
            If there was a new owner conflict for the object
            Then Add to the model path record the new path owner resulting from
                                the new owner request performed in the
                                same partition as the new owner for the object
                                and the new virtual partition name
            Else Add to the model path record the new path owner resulting from
                                the new owner request having the largest
                                associated partition tag value
                                and the new virtual partition name
            EndIf
        EndCase
    EndFor
```

Figure VI.8 - Algorithm for Resolving Conflicting Ownership Requests.

winning new owner for the object. This consistency is achieved by selecting winning usurps from the same change list as the winning new owner for the object. Each change list represents a portion of the consistent view held by one of the merging partitions; therefore, by selecting winning usurps from the same change list we are achieving a consistent view of ownership. Figure VI.8 presents the algorithm for resolving conflicting change of ownership requests.

Lemma 10: For the set of object or path ownership changes appearing in any of the change lists, the resolution algorithm for resolving ownership conflicts produces mutually consistent owner fields in the model records on every merging site.

Proof of Lemma 10: The resolution algorithm for resolving conflicting change of ownership requests selects the prevailing owner of an object by selecting the owner with the largest associated partition tag value. The prevailing owner of each path is the owner with the largest associated partition tag value, or the owner that was proposed by the same partition as the prevailing owner of the object. The partition tag values are from the totally ordered set of virtual partition names; therefore, every merging site will select the same prevailing object owner based on the ordering of partition tag values. Every merging site selects the same prevailing owner for each path in an object because the partition tag values are totally ordered and the same object owner was selected by every merging site. The prevailing owner field is copied as is from the change list to the model; therefore, all of the merging sites resolve an ownership conflict to a mutually consistent prevailing owner.

6.5.4 Permission Conflicts

The set_permission request can produce merge-able conflicts and direct conflicts. Set_permission is used to modify the general permission setting and to add or delete users from the inclusion and exclusion lists. Each entry in the inclusion or exclusion list has an associated partition tag. This tag value states in which virtual partition the entry was added, or the permission associated with this client was modified, or the entry was marked as deleted. Modifications to the inclusion and exclusion lists are merge-able. The resolution algorithm for resolving changes to the inclusion and exclusion lists adds each new permission entry to the model, unions the permitted operations of two entries for the same user, and marks the model entry as deleted if any change list claims that the entry is deleted. The results of grant requests and deny requests are merged in the same manner as inclusion and exclusion modifications resulting from a set_permission request. When incorporating the inclusion and exclusion lists into the local system directory, each entry must be treated as a distinct record and all modifiable fields are considered to have been modified.

Modifying the general permission setting is a direct conflict that is resolved by selecting the general permission setting that is most restrictive. The general permission settings from most restrictive to least restrictive are private, private-include, public-exclude, public. Figure VI.9 presents the algorithm for resolving conflicting permission requests.

Resolving Permission Conflicts

```
Build a new inclusion list as follows:
    For each entry in the inclusion list Do
        If the model already contains an inclusion entry for this user
        Then
            Union the specified operation permissions together
            If the entry is marked as deleted
            Then Mark the entry in the model as deleted
        EndIf
    EndFor
Build a new exclusion list as follows:
    For each entry in the exclusion list Do
        If the model already contains an exclusion entry for this user
        Then
            Union the specified operation permissions together
            If the entry is marked as deleted
            Then Mark the entry in the model as deleted
        EndIf
    EndFor
If only one set_permission request is reported in the change lists
Then Add the resulting general permission setting to the model
Else
    If any one of the conflicting permission settings is private
    Then Add the permission setting private to the model
    Else If any one of the conflicting permission
                settings is private-include
        Then Add the permission setting private-include to the model
        Else If any one of the conflicting permission
                settings is public-exclude
            Then Add the permission setting public-exclude to the model
            Else Add the permission setting public to the model
            EndIf
        EndIf
    EndIf
EndIf
```

Figure VI.9 - Algorithm for Resolving Conflicting Permission Requests.

The resolution of conflicting set_permission requests and grant and deny requests result in the same set of permissions on every merging site.

Lemma 11: For a set of permission changes appearing in any of the change lists, the resolution algorithm for resolving permission conflicts produces mutually consistent general permission fields and inclusion and exclusion lists in a model object or path record on every merging site.

Proof of Lemma 11: Let X be an object or path for which conflicting permission changes are reported in the change lists. We must show (a) that the general permission setting for X will be the same at every merging site and (b) that the inclusion and exclusion lists for X will be the same at every merging site.

(a) All merging sites receive the same set of change list entries. The ordering of general permission settings from most restrictive to least restrictive is a total ordering; therefore, each merging site selects the same most restrictive general permission setting from the set of general permission proposed by the change lists. This most restrictive setting is copied into the model record for X.

(b) Every merging site receives the same set of change lists; therefore, all merging sites will add the same new inclusion and exclusion entries to the model. When the inclusion or exclusion entry for a single user appears in more than one change list the permitted operations are unioned with the operations permitted in the model entry. Union is a commutative operation, so all merging sites produce the same list of permitted or restricted operations for each user entry added to the model. Finally, if any change list contains a permission entry that is marked as deleted, all merging sites will mark the model permission entry as deleted. Therefore, all merging sites construct mutually consistent model permission entries.

Therefore, the resolution of permission changes produces mutually consistent permission entries in the model records of all the merging sites.

6.5.5 Derive Conflicts

Derive requests conflict with other derive requests by creating a duplicate name or by associating the same implicit alias number with different paths. All duplicate names are managed by the merge protocol as discussed in the subsection on naming conflicts. Conflicting implicit alias numbers are manifested by two logically distinct paths with distinct immutable tokens having the same implicit alias number. Implicit alias numbers for

one object are unique within a single partition; therefore, if K partitions are merging, there may be at most K logically distinct paths with the same alias number. In such a case (K-1) of the paths must be given new unique implicit alias numbers. The (K-1) new implicit alias numbers are selected by determining the largest alias number used in an object up to this point in time and assigning increasingly larger alias numbers to each new path in a well-defined order. A well-defined order is achieved by processing the (K-1) paths in the order determined by the name of the virtual partition that proposed the path. Because implicit alias numbers are generated in increasing order per object, the largest implicit alias number used in object O can be unilaterally calculated by each site in the merge.

Lemma 12: The largest implicit alias number used in an object O is the maximum of

(a) the largest implicit alias number appearing in the change list records about paths in object O,

and

(b) the largest implicit alias number used in object O according to the local system directory.

Proof of Lemma 12:

Case I: If a (derive, derive) conflict exists then two or more of the virtual partitions participating in the merge have created one or more new paths in object O during their most recent virtual partition membership. These new path records will contain a partition tag whose value is in the tag set used by the merge. A copy of each of the new path records will be included in the change lists exchanged among the merging partitions. Each new path created was assigned an implicit alias larger than the largest alias number for object O known in the creating partition. The virtual partition that has created the most paths in object O during its complete partition history will have assigned the largest implicit alias number to a new path. Thus, the largest implicit alias number used in object O by any partition must appear in the change list proposed by one of the partitions.

Case II: If no (derive, derive) conflict exists, then at most one of the merging partitions created zero or more new paths in the object O while the sites were partitioned. If no new paths were created then the local system directory on every site contains information on all existing paths in object O, thus the largest implicit alias number used in object O is in the local system directory at each merging site. If one partition created new paths in object O, the new path records contain a partition tag

whose value is in the tag set used by the merge and a copy of the new path records will be included in that partition's change list. Each merging site will obtain a copy of the change list. Thus, the largest implicit alias number used in object O must appear in the change lists processed by the merging sites.

Resolving Duplicate Implicit Alias Numbers

```
Determine the maximum implicit alias number used in object O, call it A_e
If there are any duplicate alias numbers associated with distinct model paths
Then
    Determine the maximum alias number not duplicated in any of the change
            list records for object O, call it A_b
    Let MaxAlias = A_e
    For A = (A_b + 1) to A_e Do
        Let P be the set of distinct model paths claiming A as their alias number
        Order the set P in descending order by the name of the
                partition that created the path
        For the first model path in the set P set the alias number to be A
        Set the associate partition tag field to the name of the
                newly formed virtual partition
        For the remaining paths in the ordered set P Do
            Increment MaxAlias
            In the model record for this path, set the alias number to MaxAlias
            Set the associate partition tag field to the name of the
                    newly formed virtual partition
        EndFor
    EndFor
EndIf
```

Figure VI.10 - Algorithm for Resolving Duplicate Implicit Alias Numbers.

When a site has calculated the largest implicit alias number used for an object O, all of the model paths that cannot retain the alias number originally assigned will be ordered by partition name and assigned unique alias numbers. Figure VI.10 presents the algorithm for resolving duplicate alias numbers. The assignment of new implicit alias numbers is done in a well-defined order making the assignment mutually consistent on the merging sites.

Lemma 13: The algorithm for resolving duplicate implicit alias numbers ensures that each logically distinct path in an object O is associated with a unique implicit alias number and these alias numbers are assigned to the model path records in a mutually consistent manner by each of the merging sites.

Proof of Lemma 13: Let P_1, P_2, ..., P_N be N merging virtual partition in a federation F. Let O be an object in federation F. We must demonstrate two things: (a) after executing the duplicate alias resolution algorithm, each logically distinct path in an object O is associated with a unique implicit alias number; and (b) after executing the duplicate alias resolution algorithm, each of the merging sites associates the same implicit alias number with a given distinct model path in an object O.

(a) Let p1 and p2 be two distinct paths in an object O. Let t_{p1} and t_{p2} be the unique path tokens associated with paths p1 and p2 respectively. Assume that after the duplicate alias resolution algorithm has been executed, p1 and p2 are both associated with the implicit alias number A.

Case i: Assume that path p1 and p2 were assigned their alias number by the resolution algorithm. The resolution algorithm assigns the alias numbers $(A_b + 1)$, $(A_b + 2)$, ..., A_e, $(A_e + 1)$, ... Clearly all alias numbers assigned by the resolution algorithm are unique. But, paths p1 and p2 were assigned the same alias number. Contradiction.

Case ii: Assume that path p1 and path p2 retained the alias number assigned during normal processing. Suppose path p1 and path p2 were created in one virtual partition P_i. This represents an internal inconsistency in partition P_i. By corollary 3, internal consistency is maintained in a single virtual partition. Contradiction. Therefore, path p1 and path p2 were created in distinct virtual partitions P_i and P_j respectively. If partitions P_i and P_j are two of the merging partitions, partition P_i must include in its change list the record for path p1 because path p1 was created without the knowledge of partition P_j. Similarly, partition P_j must include in its change list the record for path p2 because path p2 was created without the knowledge of partition P_i. If path p1 and p2 were both reported in change lists for the merge, the fact that two distinct path tokens t_{p1} and t_{p2} have the same implicit alias number will be detected by the resolution algorithm and the paths p1 and p2 will be assigned unique alias numbers. Contradiction.

Case iii: Assume that path p1 was assigned a new implicit alias number by the duplicate alias resolution algorithm and path p2 retained the alias number assigned during normal processing. The resolution algorithm must assign to path p1 an alias number greater than A_b. The alias number (A_b + 1) was the smallest alias number to be duplicated; therefore, (A_b + 1) is associated with a set of paths that were created without the knowledge of one of the merging partitions. Because paths p1 and p2 have the same alias number, path p2 must have been created without the knowledge of one of the merging partitions; therefore, path p2 must be reported in the change list of the partition that created path p2. Then path p1 and p2 are both reported in change lists for the merge, then by case i, path p1 and path p2 would have been assigned unique alias numbers. Contradiction.

(b) The order in which the unique implicit alias numbers are assigned to distinct paths is determined by the name of the virtual partition that proposed the original alias number for that path. The space of virtual partition names is totally ordered; therefore, the assigning of unique implicit alias numbers by the resolution algorithm is a totally ordered process. All merging sites follow the same total ordering; therefore all merging sites assign unique implicit alias numbers in a mutually consistent fashion.

6.5.6 Deletion Conflicts and Physical Deletion

The delete request and the erase_all request may conflict with almost all of the other requests. All deletion conflicts are classified as merge-able conflicts except when deletion conflicts with update. As discussed in chapter three, when a deletion request is executed, the affected instances are logically rather than physically removed. When a change list entry states that a system directory record has been logically deleted, that logical deletion is always propagated to the corresponding model record unless the deletion conflicts with an update request. When one partition updates a path that has been deleted in another partition, the update must be saved. If an update request follows a delete request in the same partition the update will be failed as discussed in chapter three. This is acceptable, but once the update has been accepted by one partition it may not be rejected regardless of the reason. This is in keeping with the general system philosophy of "never throw away user data". Saving the update requires that some or all of the deleted instances be reincarnated. If an

update has been performed by one of the merging partitions these records will be reincarnated and the updates will be added according to the resolution algorithm for resolving (update, update) conflicts.

When an object or path is logically deleted, the records that describe the object or path will be processed by merge recovery as if they were not deleted; that is, the merge and resolution algorithms for name conflicts, assign conflicts, ownership conflicts, permission conflicts, and derive conflicts are applied to the logically deleted records, but the records remain logically deleted. The state of being deleted is merged with the results of other requests so that the records that describe the deleted object or path remain mutually consistent among the communicating sites. We wish to maintain the mutual consistency of these records in the event that a conflicting update request may be encountered in a future merge recovery, forcing the reincarnation of the deleted entity. When the entity is reincarnated, the reincarnation must be mutually consistent on all merging sites.

All of the instances affected by a delete or erase_all may be physically removed when it has been ascertained that there will be no need to reincarnate any of these instances in a future merge recovery. Physical deletion may take place only after all sites in a federation have merged to form one virtual partition, and all sites know that every other site has committed to that new virtual partition. The algorithm for determining when and if this state is achieved is presented in figure VI.11.

The algorithm must be initiated at the end of a merge recovery. The required condition of full membership in the forming virtual partition is easily tested by examining the partition commit message. If the commit message specifies that full membership should result from the merger, then each site determines that every other site has actually completed the merge recovery and committed to the new unanimous partition by keeping track of the sites from which multicast messages have been received. Once a site S has sent a multicast message and has received a multicast message from every other site, the site S may physically delete all of the logically deleted data. During the existence of the virtual partition with full membership, all client requests for deletions must be processed as logical deletions rather than physical deletions. The request processing protocol uses one-phase commit for sending multicast messages. If a site or communication failure occurred during the multicast sending of deletion results, the sites that received the message would perform physical deletion. The sites that did not receive the message could form new virtual partitions and honor requests

Algorithm for Physcally Deleting Logically Deleted Data

```
If the partition commit message specifies that all sites in a federation F
                    have agreed to join the new virtual partition
Then
    If a new recovery begins
    Then
        For each entry in the multicast message log Do
            Unmark the entry for being received after a full merge
        EndFor
        Exit
    Else
        For each multicast message received Do
            Perform normal processing of the multicast message
            Mark this multicast log entry as received after full merge
            If the multicast log entry for each site is marked as
                        received after full merge and the local site has
                        sent a multicast message
            Then Perform physical deletion on deleted entities
            EndIf
        EndFor
    EndIf
EndIf
```

Figure VI.11 - Algorithm for Physically Deleting Logically Deleted Data.

that necessitate the reincarnation of the physically deleted data during a future merge recovery. The algorithm for determining when and if physical deletion is allowed determines that it is acceptable to physically delete everything that has been logically deleted prior to the formation of the new virtual partition; the algorithm does not allow us to draw conclusions about future deletions.

When an update request conflicts with an erase_all request, the update must be saved. In this case it is possible to honor the logical erase and save the update by placing the new instance formed by the update in a new alternate path derived from the path that has been erased. All instances that are common between the old erased path and the newly derived path must be reincarnated. The resolution algorithm for resolving (update, update) conflicts (to be discussed in the next section) will automatically form new alternate version paths for the instances created by losing update requests. We make use of this facility by designating the conflicting update request to be a losing update. By doing this we are assured that the new instance created by the update will be placed in a new alternate version path by the update resolution algorithm. The update resolution algorithm will use the group update control information at the beginning of the change list to select winning and losing updates.

Designating the update as a losing update means that we must locate this update in the control information we have saved and mark the update as a losing update. The erase request is honored by marking the model path record as deleted. Figure VI.12 summarizes the algorithm for resolving conflicts between update request and erase_all requests.

Resolving Conflicting Deletions and Updates

If the conflict is (erase all, update)
Then
    Locate this update in the control information for group updates
    Mark the update and all of its successor updates as losing updates
    Mark the model path record as deleted
EndIf
If the conflict is (delete, update)
Then
    Mark the model object record as undeleted
    When incorporating the object record *Do*
        If the object record exists locally and is logically deleted
        Then Undelete the object record
        Else Add the object record to the local system directory
    EndWhen
EndIf

Figure VI.12 - Algorithm for Resolving Conflicting Update and Deletion Requests.

When one partition performs an update on any path in an object that has been deleted in another partition, the update must be saved. Whether or not the update will be selected as a winning update or a losing update, all instances of the object must be reincarnated. We reincarnate the entire object rather than just the path affected by the update to ensure that every object has a principal version path. An object without a principal version path could result when a conflicting update request has updated a non-principal path. Reincarnating the object means that all records describing the object must be added to the model and must be marked as undeleted. When the object record is incorporated into the local system directory, if the local directory contains the object record, the deletion indicator must be removed. The result of the update will be added to the model by the algorithm for resolving conflicting updates. Figure VI.12 presents the algorithm for resolving delete or erase_all requests that conflict with update requests.

To demonstrate that this algorithm is feasible, we must show that no data is physically deleted prematurely.

Lemma 14: When all instances in an object or path have been physically deleted, a future merge will never attempt to reincarnate those instances.

Proof of Lemma 14: Let S_1, S_2, ..., S_n be all of the sites belonging to a federation F. Let X be a collection of data instances representing an object or path in the federation F. Assume that all instances in X have been physically deleted. A merge recovery will attempt to reincarnate X if one or more of the merging partitions has processed an update request on X. If one partition processed an update request on X there exists a site in that partition which initiated the request. Let S_i be the site that initiated the update request on X. By the definition of normal processing of an update request, if S_i processed the update request on X then X was not logically deleted or physically deleted on site S_i. Contradiction.

If we assume that the resolution algorithm resolves (update, update) conflicts in a mutually consistent manner, then to show that the resolution of conflicting update and deletion requests is managed in a mutually consistent manner we must show that when an object O is reincarnated, the resulting object is mutually consistent on every merging site.

Lemma 15: If the resolution algorithm selects an object O for reincarnation, the reincarnated object O will be mutually consistent on every merging site.

Proof of Lemma 15: Let O be an object that the resolution algorithm wishes to reincarnate. By the contrapositive of lemma 14, we know that O is not physically deleted on any of the merging sites. Consider the process of reincarnation by a merging site S. For each system directory record that describes any part of the object O, site S either has a logically deleted copy of the record in the local system directory or site S does not have a copy of the record. If site S has a copy of the logically deleted record, that record can be altered only by a merge recovery. By lemma 7, lemma 8, lemma 9, lemma 10, lemma 11, and lemma 13 the local copy must be mutually consistent with every other site that has a local copy of the record. If site S does not have a copy of the logically deleted record, then that record must have been created and logically deleted in response to requests that were initiated in virtual partitions of which site S has no knowledge. The record must contain a partition tag associated with the creation and the logical deletion of the record. These partition tag fields must have a value in the range of interest; therefore, a copy of the record must

appear in at least one of the change lists. Site S will copy the record, as it appears, into the model. By the above argument, all sites that had a local copy of the record had a mutually consistent copy; one of these sites has provided a copy of the record for inclusion in a change list. Therefore, those sites without a local copy of the record receive the same mutually consistent version of the record. Lemma 7 ensures that the mutually consistent model record becomes a mutually consistent directory record. Each reincarnated record describing an object will be mutually consistent among all of the merging sites; therefore, a reincarnated object O will be mutually consistent among all of the merging sites.

Lemma 16: The resolution algorithm for resolving delete or erase_all requests that conflict with update requests maintains mutual consistency among the merging sites.

Proof of Lemma 16: Let R be a delete or erase_all request performed on an object O. Let U be an update request that adds a new instance I to a path P in an object O. By definition of the resolution algorithm, if R is an erase_all request, some or all of the instances in the path P are reincarnated, a new alternate path is created, and the new instance I is added to the object O as a losing update; if R is a delete request, the entire object O is reincarnated and the new instance I is added to the object O according to the algorithm for resolving (update, update) conflicts. By lemma 15, the reincarnation of path P or the reincarnation of object O is mutually consistent among the merging sites. By lemma 20 (presented in the next section), the new instance records added to the model of object O, due to the update requests, are mutually consistent on the merging sites. Lemma 7 ensures that these mutually consistent instance records will be added to each local system directory in a mutually consistent manner; therefore, the algorithm for resolving conflicting deletion and update requests produces a mutually consistent object O on every merging site.

6.6 Resolving Group Update Conflicts

A single virtual partition manages conflicting updates within that partition by the creation of alternate versions. When multiple virtual partitions merge, update conflicts may arise between updates performed in separate partitions. The resolution algorithm manages group update conflicts by selecting a collection of non-conflicting group updates as "winning" updates and resolving conflicting updates by creating alternate versions.

The selection of winning updates is based on the selection of non-conflicting updates. However, the state of being non-conflicting is not sufficient for selection as a winning update. The selection of winning group updates is controlled by four requirements:

1. A winning group update must not conflict with any other winning group update.

2. Every instance created by a winning group update is a winning instance and will remain in the version path it was placed in by the update.

3. If an instance I, in a version path P, is created by a winning group update G, then every instance that is a predecessor of instance I in path P must be the result of a winning update.

4. If an instance I, in a version path P, is created by a losing group update G executed, then every instance that is a successor of instance I must be the result of a losing update.

The first condition requires that for each path, only updates from one partition may be selected as winning updates for that path. The second condition requires that all instances resulting from a winning group update will prevail as winning instances. To support this requirement the system must maintain information on the group update relationship among all instances. When a group update is executed by the storage system, each group update is assigned a unique immutable token. Each instance record in the system directory will record the token corresponding to the update that created the instance. The third and fourth conditions require that a sequence of updates to a path performed by one partition must begin with zero or more winning updates followed by zero or more losing updates. The winning and losing updates may not be interspersed in such a sequence.

When a group update G is selected as a winning update, every update that conflicts with G must be a losing update. In an attempt to minimize the amount of change brought about by the resolution procedure, we will attempt to maximize the number of winning updates. There are at least two ways to measure the number of winning updates. Consider the following model of group updates.

Let $P_1, P_2, ..., P_N$ be virtual partitions involved in a merge.

Let $G_{i,1}, G_{i,2}, ..., G_{i,n}$ be group updates executed in virtual partition P_i.

Let $x_{i,j,1}, x_{i,j,2}, ..., x_{i,j,m_{i,j}}$ be the version paths updated by the group update $G_{i,j}$.

Two group updates $G_{i,j}$ and $G_{k,l}$ conflict iff $x_{i,j,r} = x_{k,l,s}$ for some $r, 1 \le r \le m_{i,j}$ and some $s, 1 \le s \le m_{k,l}$ and $i \neq k$. The conflict relationship is symmetric; that is, if A conflicts with B, then B conflicts with A. There are no conflicts among the set of updates executed in a single partition.

Define the set of winners

$$W = \left\{ G_{i,j} \;\middle|\; \begin{array}{l} \text{if } G_{i,k} \textit{ created an instance that is} \\ \quad \textit{a predecessor of an instance created by} \\ \quad G_{i,j} \textit{ then } G_{i,k} \in W \\ \text{and if } G_{i,j} \in W \text{ and } G_{r,s} \in W, \\ \quad G_{i,j} \textit{ does not conflict with } G_{r,s} \end{array} \right\}$$

Two possible criteria for selecting a maximum number of winning group updates are:

1.

$$\textit{maximize} \quad \sum_{\{i,j \mid G_{i,j} \in W\}} m_{i,j}$$

2.

$$\textit{maximize} \quad |W|$$

Criteria one maximizes the number of winning version paths from the group updates. Criteria two maximizes the number of winning group update requests. These two criteria are not equivalent. Consider the example illustrated in figure VI.13. Three virtual partitions P_1, P_2, and P_3 performed the group updates $G_{1,1}$, $G_{2,1}$, $G_{2,2}$, and $G_{3,1}$. Update $G_{1,1}$ conflicts with update $G_{2,1}$ and update $G_{1,1}$ conflicts with update $G_{2,2}$. The set $Winners_1$ is the set of

winning group updates according to criteria 1. The set $Winners_2$ is the set of winning group updates according to criteria 2.

Let P_1, P_2, and P_3 be three virtual partitions.

Let x_1, x_2, x_3, x_4, x_5, x_6, x_7, and x_8 be objects known in the three partitions.

The group updates executed in these partitions were

$G_{1,1}$ = update x_1, x_2, x_3, x_4, x_5
$G_{2,1}$ = update x_1, x_2
$G_{2,2}$ = update x_5, x_6
$G_{3,1}$ = update x_7, x_8

The conflict sets are

$G_{1,1}$ conflicts with { $G_{2,1}$, $G_{2,2}$ }
$G_{2,1}$ conflicts with { $G_{1,1}$ }
$G_{2,2}$ conflicts with { $G_{1,1}$ }
$G_{3,1}$ conflicts with { }

The two solution sets are

$Winners_1$ = { $G_{1,1}$, $G_{3,1}$ }
$Winners_2$ = { $G_{2,1}$, $G_{2,2}$, $G_{3,1}$ }

Figure VI.13 - Comparing Two Maximization Criteria for Selecting Group Updates.

Maximization by either criteria can be expressed as a problem in integer programming. We define the problem of selecting a maximal collection of winning group updates as follows.

Let P be the set of merging virtual partitions P = { P_i | $1 \le i \le I$ }.

Let X be the set of version paths known by any of the merging partitions

X = { x_j | $1 \le j \le J$ }.

Let G be the set of group updates G = { G_k | $1 \le k \le K$ and each G_k = { x_j | $x_j \in X$ } }.

Define the function f:(P,X) → {G, ∅} to map a virtual partition and a version path to the set of group updates performed on that path by sites in that partition.

$$f(P_i, x_j) = \left\{ G_k \mid x_j \in G_k \text{ and } G_k \textit{ was executed in partition } P_i \right\}$$

Define the function pred:(P,X,G) → {G, ∅} to map an instance to the set of group updates that created the immediate predecessor instances created by any of the merging partitions.

$$pred(P_i, x_j, G_k) = \left\{ G_l \mid \begin{array}{l} G_l \text{ created an instance of } x_j \text{ that is a} \\ \text{predecessor of the instance of } x_j \text{ created by } G_k \\ \text{and reported by partition } P_i \end{array} \right\}$$

The function f, from the domain of partitions cross version paths to the range of group updates, is a one-to-many function. In a single partition P, a version path x may have been updated by several group updates. The function pred, from the domain of partitions cross version paths cross group updates to the range of group updates, is a one-to-many function. In a single partition, each instance has zero or more predecessor instances and each of those instances was created by a distinct group update.

To maximize the number of winning update version paths (meeting criteria one), we define a selector function s1:(P,X,G) $\rightarrow$ {1,0} to select the winning paths. The selector function s1 is one-to-one.

The goal is

Maximize $\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} s1_{ijk}$ subject to

$$s1_{ijk} = \begin{cases} 1 \text{ if } f(P_i, x_j) \supset G_k \text{ and} \\ \quad a)\ s1_{ajl} = 0 \ \forall\, a \neq i \text{ and } \forall\, l \neq k \\ \quad b)\ s1_{ijl} = 1 \text{ if } G_l \subset pred(P_i, x_j, G_k) \\ \quad c)\ s1_{ibk} = 1 \text{ if } f(P_i, x_b) \supset G_k \\ 0 \ \textit{otherwise} \end{cases}$$

To maximize the number of winning group update requests (meeting criteria two) we define a selector function s2:(G) $\rightarrow$ {1,0} to select the winning update requests. The selector function s2 is one-to-one.

The goal is

Maximize $\sum_{i=1}^{I} s2_i$ subject to

$$s2_i = \begin{cases} 1 \text{ if } f(P_j, x_k) \supset G_i \text{ and} \\ \quad a)\ s2_l = 0 \ \forall\, G_l \subset f(P_a, x_k) \textit{ where } a \neq j \\ \quad b)\ s2_l = 1 \ \forall\, G_l \subset pred(P_j, x_k, G_i) \\ 0 \ \textit{otherwise} \end{cases}$$

The problem of integer programming was shown to be NP-Complete by Karp [46]. We propose an approximate solution to the problem, using heuristics to select a large set of conflicting group updates. Our heuristics allow us to assign a "goodness" value to each of the group updates to be merged. The set of group updates is ordered by their associated goodness value. A linear greedy algorithm scans the updates in order and selects collections of related non-conflicting group updates as the winning updates.

Our heuristic for ordering the set of group updates is a combination of criteria one (to maximize the number of winning version paths) and criteria two (to maximize the number of winning update requests). We measure the effect of a group update G on the number of winning version paths by counting the number of version paths updated by G. This factor estimates the user time cost associated with an update. We assume that the number of versions updated in one request is directly proportional to the user effort expended in preparing the data to be updated. If an update results in the creation of alternate versions, the user may perform additional work to manage or merge those versions. Giving preference to group updates with greater breadth should reduce the additional user effort required in managing undesired alternate versions. We measure the effect of an update G on the number of winning update requests by counting the number of updates that are predecessors to G. If update G is selected as a winning update, all of the predecessors to G must also be winning updates; thus, we are giving preference to longer chains of updates. These longer update chains reflect more update activity for these versions. We assume that more update activity means that the end-users have invested a larger amount of effort than was expended on versions with fewer updates. Giving preference to the longer chains of updates is again an attempt to conserve user effort by selecting those updates that may be associated with a large user time cost. When calculating a goodness value for a group update we give preference to criteria one over criteria two by doubling the count corresponding to criteria one before adding the count contributed by criteria two. We calculate the goodness value of a group update G as twice the number of version paths updated by G plus the number of predecessor updates for G that are reported by any of the merging partitions.

The number of version paths updated by one group update is information that is readily available in the group update control information of the change list. In order to count the number of predecessor updates for a group update G, we must first construct the set of predecessor updates. The set of predecessor updates is also used in the greedy algorithm to select the set of winning updates. The predecessor sets are calculated from the group update

control information contained in the change list. Figure VI.14 presents the algorithm for determining the set of predecessor updates for a given group update G. The predecessor updates are those updates that created instances which are predecessors to the instances created by group update G. Each instance reported in the change list control information reports its immediate predecessor instance. Using the information on immediate predecessors, we construct an ancestral instance chain dating back to the last ancestral version reported by any of the merging partitions. The group updates that created these instances are the set of predecessor updates.

Calculate Predecessor Set for a group update G

Predecessor Set for a group update G is

$$\bigcup_{j=1}^{N} Pred\_Updates(I_j) \text{ where } G = update\ I_1, I_2, ..., I_N$$

Calculate Predecessor Updates for an instance I in a path X

Consider the group update control information from all of the merging partitions
Constructed an ordered set Pred as follows
    Initialize Pred to the instance I created by $G_{i,j}$ in path X
    Repeat
        If the predecessor of the last instance in the Pred set is reported by any merging partition
        Then Add that predecessor as the new last member of the Pred set
    Until the size of the Pred set does not increase
Set Pred_Updates for instance I to be the empty set
For each instance in the Pred set *Do*
    Pred_Updates = Pred_Updates ∪ the group update that created this instance
EndFor

Figure VI.14 - Algorithm for Calculating the Predecessor Updates for a Group Update.

Once the set of predecessor updates have been calculated, we compute a goodness value for each group update and order the updates by goodness value. If the goodness value is not unique, the name of the partition reporting the update is used as a secondary sort key. If the goodness value and the name of the reporting partition are not unique, then the group update token associated with the update is used as a ternary sort key. Each group update token is globally unique; therefore, the set of group updates form a total ordering. This total ordering determines the order in which the updates will be processed when selecting winning group updates.

Figure VI.15 presents our algorithm for selecting winning group updates. The algorithm uses the conflict relationship defined earlier in this chapter. (Two group updates conflict if the two updates were executed in different partitions and the set of version paths updated by each has a non-empty intersection.) We use the set of predecessor updates to select the set of related winning updates and the set of successor updates to select the set of related losing updates. For a group update G, the predecessor updates are calculated by our previous algorithm. The set of successor updates for G are those updates, that created one or more instances which are successors to the instances created by. The successor instances are reported in the same change list as G and are restricted to those instances belonging to the same version path as the instance created by G. A group update is selected as a winning group update only if all of its predecessor updates do not conflict with any of the previously selected winning updates.

The fact that this algorithm yields only an approximation to the true maximal solution is seen by considering three group updates $G_{1,1}$, $G_{2,1}$, and $G_{3,1}$, with goodness values 7, 5, and 4 respectively. Suppose $G_{1,1}$ conflicts with $G_{2,1}$ and $G_{3,1}$, but $G_{2,1}$ does not conflict with $G_{3,1}$. Our algorithm would begin by selecting $G_{1,1}$ as a winning update. $G_{2,1}$ and $G_{3,1}$ cannot be selected as winning updates, so our total goodness value for winning updates is 7 and the total goodness value for losing updates is $(5 + 4) = 9$. The true maximal solution is to select $G_{2,1}$ and $G_{3,1}$ as winning updates.

Lemma 17: The update selection algorithm selects a mutually consistent set of updates at every merging site.

Proof of Lemma 17: Each merging site receives the same change list information. The set of all group updates form a total ordering based on the assigned goodness value, the name of the virtual partition reporting the update, and the update token associated with the update. Every merging site computes the same ordering of group updates based on the total order; therefore, every merging site processes the group updates in the same order. For each group update G, every merging site calculates the same set of predecessor updates because each site receives the same predecessor information in the change lists. Each site uses the same test for conflicting updates applied to the totally ordered set of all group updates; therefore, every site must select the same set of winning group updates.

Algorithm for Selecting Winning Group Updates

Let group update $G_{i,j}$ be the j-th update in the change list of partition P_i
Define PredUpdate($G_{i,j}$) = { $G_{a,b}$ | $G_{a,b}$ created a predecessor instance
for one or more of the instances created by $G_{i,j}$
and P_a is one of the merging partitions }
Define SuccUpd($G_{i,j}$) = { $G_{i,k}$ | $G_{i,k}$ created a successor instance
for one or more of the instances created by $G_{i,j}$
and reported by P_i as an instance of the same version path }
Associate with each group update $G_{i,j}$ a count
For each $G_{i,j}$ *Do*
Set $G_{i,j}$'s associated count to two times the number of objects updated by $G_{i,j}$
plus the number of updates in PredUpdate($G_{i,j}$)
EndFor
Sort the $G_{i,j}$'s in descending order by their associated counts,
use the virtual partition name as a second key,
use the group update token as a third key
Winners = { }
Losers = { }
For each $G_{i,j}$ in descending order *Do*
If the current $G_{i,j}$ and all updates in PredUpdate($G_{i,j}$)
do not conflict with any $G_{r,s}$ ($i \neq r$) in Winners
Then
Mark the control information for $G_{i,j}$ and PredUpdate($G_{i,j}$)
to be winning updates
Remove $G_{i,j}$ and PredUpdate($G_{i,j}$) from the sorted update list
Else
Mark the control information for $G_{i,j}$ and SuccUpd($G_{i,j}$)
to be losing updates
Remove $G_{i,j}$ and SuccUpd($G_{i,j}$) from the sorted update list
EndIf
EndFor

Figure VI.15 - Algorithm for Selecting Winning Group Updates.

6.7 Processing Losing Group Updates

Creating alternate version paths for instances created by losing group updates poses three problems. First, if one partition performed K updates on a path and the L most recent of these updates ($K \geq L$, $L > 1$) were declared to be losing updates, then the corresponding L losing instances should be placed in only one, rather than L, new alternate version paths. Second, each site performing the merge processing must unilaterally create the same name and assign the same implicit alias number to a newly created path. Third, creation of alternate paths must support configuration write consistency.

6.7.1 Grouping Losing Update Instances

For each path in an object, at most one new alternate path may be created per partition involved in the merge. Basically this says that if K partitions propose a set of instances as the continuation for a path, then at most K new alternate paths need to be created (one per partition). The problem is solved by grouping temporally related instances created by a single partition. We form an association among the temporally related losing versions from a single partition and place then in one alternate path.

6.7.2 Unilateral Generation of Implicit Alias Numbers and Tokens

Our second problem is how to create the same alternate paths unilaterally on each site. Two paths are the same path if they have the same implicit alias number and the same immutable token. Thus, we must solve two subproblems. First, each site must assign the same implicit alias number to the same logical path. The assigned implicit alias numbers must be unique in the set of implicit alias number for the affected object and must be generated in increasing order. Second, each site must assign the same immutable token to the same logical path. The assigned token must be unique in the space of all tokens.

Each site is able to assign the same implicit alias number to a new path by first, determining the largest alias number used in an object up to this point in the merge algorithm and second, assigning increasing alias numbers to each new path in a well-defined order. In our discussion of derive conflicts, we showed in lemma 12 that the largest implicit alias number used in an object O is unilaterally computable from the local system directory and the change lists.

Lemma 18: After resolving all conflicting derive requests, the largest implicit alias number used in an object O is the maximum of

(a) the largest implicit alias number appearing in the change list records about paths in object O,

(b) the largest implicit alias number used in object O according to the local system directory,

(c) the largest implicit alias number used when resolving derive conflicts in object O.

Proof of Lemma 18:

Case I: If there are no derive conflicts for object O, we define the largest implicit alias number used to resolve derive conflicts in object O to be zero. All implicit alias numbers assigned by the storage system during normal processing or recovery processing must be greater than zero. By lemma 12, every site can correctly compute the maximum implicit alias number used in an object O prior to the resolution of derive conflicts. Given that no derive requests were resolved, the maximum implicit alias number used after the resolution of zero derive conflicts is the same as the maximum alias number used prior to the resolution of the derive requests.

Case II: If there are derive conflicts among new paths in object O, these derive conflicts are resolved before the update conflicts are resolved. By lemma 12, the determination of the largest implicit alias number used in object O for resolving derive conflicts is correctly calculated by every merging site. By lemma 13, every merging site assigns the same alias number to each new path created when resolving derive conflicts; therefore, the sites must use the same number of new implicit alias numbers to resolve the derive conflicts for the object O. All sites have used the same largest implicit alias number for the object O while resolving derive conflicts. The largest implicit alias number used following the resolution of derive conflicts must be greater than the largest implicit alias number used prior to resolving the derive conflicts; therefore, all sites calculate the same largest implicit alias number used in an object O.

A well-defined order for assigning the new alias numbers is achieved by ordering the paths, first by their immutable tokens and second by the virtual partition that proposed the path. By this method, each new alternate path is assigned the same implicit alias number by every merging site.

Lemma 19: The algorithm for assigning implicit alias numbers to the new alternate paths created when resolving update conflicts results in the assignment of mutually consistent implicit alias numbers at all merging sites.

Proof of Lemma 19: By lemma 18, all merging sites calculate the same largest implicit alias number used in an object O. Let A_{max} be the calculated largest alias number. All sites generate new alias numbers $(A_{max} + 1)$, $(A_{max} + 2)$, ... as they are needed. All merging sites process the paths of object O in order by the immutable token associated with the path. For each path, each merging partition will cause the creation of at most one new alternate path. These new alternate paths are created in order by the name of the virtual partition that proposed the set of instances which comprise that version of the path. Virtual partition names are tokens. The set of all tokens is totally ordered; therefore, ordering the new alternate paths first by path token and second by the name of the proposing virtual partition creates a total ordering on the new alternate paths. All merging sites assign the alias numbers $(A_{max} + 1)$, $(A_{max} + 2)$, ... in this totally defined order; therefore, the assignment of implicit alias numbers to new alternate paths is mutually consistent among all merging sites.

Each new alternate path is assigned a unique immutable token by preallocating a token for the alternate path. The preallocation is performed as part of the normal update processing. When an update is executed an immutable token is generated to represent the new instance. A second token is generated and associated with the new instance. If the update which created this new instance is found to be in conflict with another update at merge time then the second token associated with the losing instance is used as the token for the new alternate path. A new alternate path may contain several instances. The token for the new path is the second token associated with the current instance on that path. Thus, each site participating in the merge assigns an implicit alias number and an immutable token to a new alternate path in a mutually consistent manner.

6.7.3 Maintaining Configuration Write Consistency

During normal processing, the storage system supports configuration write consistency in group updates by applying a set of group update atomicity rules to determine which updates within the group will be processed normally and which updates will result in the creation of alternate versions. When the storage system resolves update conflicts in a merge recovery by demoting some of the updates to alternate versions, the demotion process must maintain configuration write consistency by applying the same group update rules used during normal processing. Additional information is required to properly apply the atomicity rules during a merge recovery. In particular, we must know the status of the set of items at the time of the original processing of the group update. This is accomplished by recording in each instance record which group update rule was applied during the original execution of the update. This information is part of the control information on group updates and allows the resolution algorithm to determine whether or not all of the items in a group must be demoted to alternate versions. If the original update produced only new current versions for principal paths, (i.e. rule one for configuration write consistency was applied), then all of the items updated by this group update must be placed in new alternate paths.

If any other rule for configuration write consistency was applied then it may be possible to retain parts of the group update as successful updates. Figure VI.16 shows a merge of two group updates $G_{1,1}$ and $G_{2,1}$ performed in partition P_1 and P_2 respectively. We assume that these are the only updates in the merge. Update $G_{1,1}$ has been selected as a winning update, forcing update $G_{2,1}$ to be a losing update. If update $G_{2,1}$ was not executed under the constraints of configuration write consistency rule number one, then the new instance of x_3 created by $G_{2,1}$ may remain in the path x_3. Because $G_{1,1}$ is the winning update, the new instance for x_1 supplied by update $G_{2,1}$ must be placed in a new alternate path a_1. Allowing update $G_{2,1}$ to be a partially successful update is possible only because $G_{2,1}$ was not a type one group update and there is no other partition proposing a different set of instances as the continuation of path x_3.

In figure VI.17, we add, to the merge, the group update $G_{3,1}$ performed in partition P_3. Again update $G_{1,1}$ is selected as a winning update and updates $G_{2,1}$ and $G_{3,1}$ are both losing updates. The new instance for x_1 supplied by update $G_{2,1}$ must be placed in a new alternate

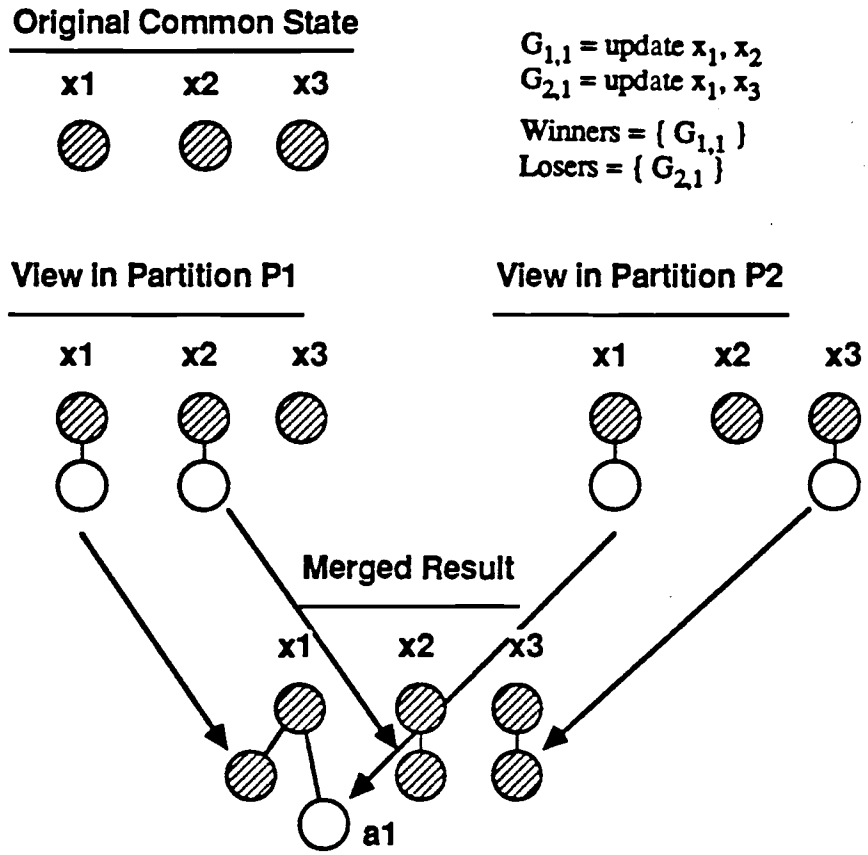


Figure VI.16 - Partially Successful Update Maintains Configuration Write Consistency.

path a_1. The new instance for x_2 supplied by update $G_{3,1}$ must be placed in a new alternate path a_2. The updates $G_{2,1}$ and $G_{3,1}$ are in conflict for achieving partial update success for path x_3. Only one of these two updates may be selected for partial update success on path x_3. Because this is a direct conflict, any unilaterally computable algorithm may be used to select the update that will be partially successful. We select the update proposed by the virtual partition with the largest name as the partially successful update. This produces the merge and resolution shown in figure VI.17.

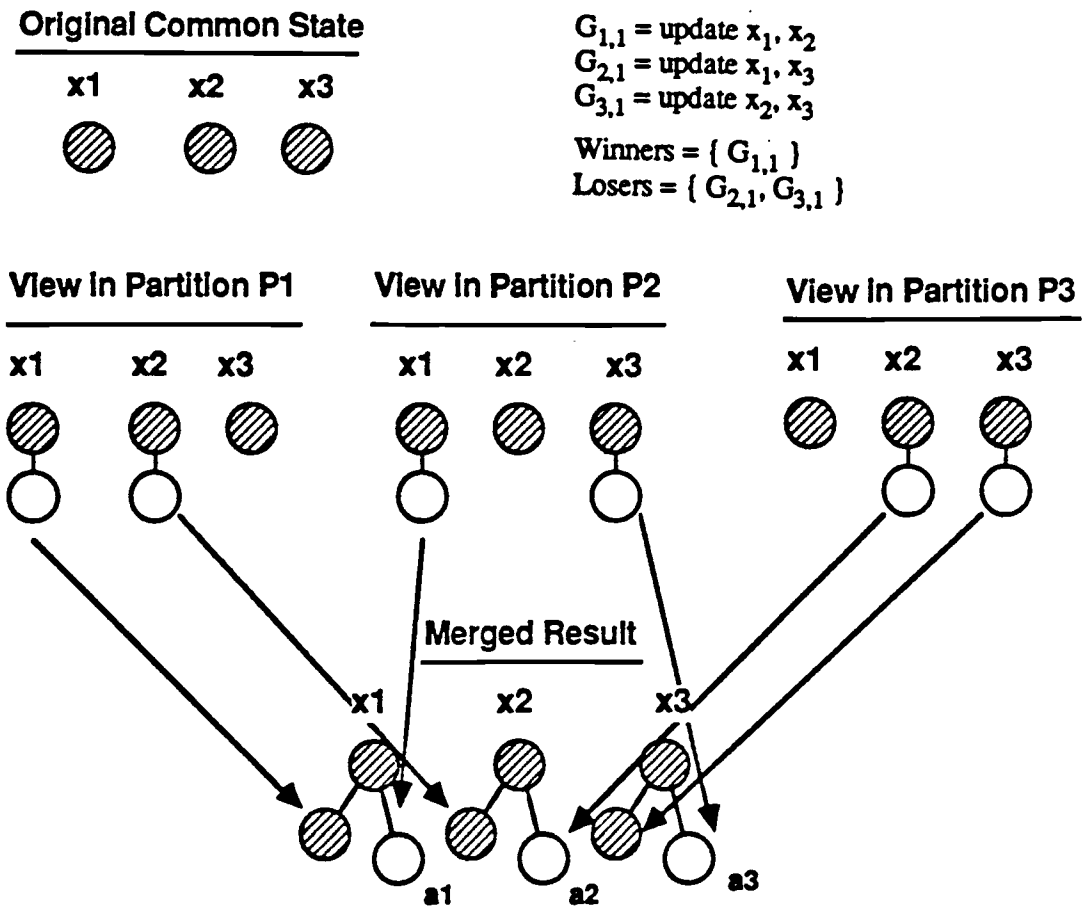


Figure VI.17 - Partially Successful Update Selection in a Merge Recovery.

Proposition 6: The resolution algorithm for resolving conflicting update requests maintains configuration write consistency.

Proof of Proposition 6: The rules for maintaining configuration write consistency require that a group update create new instances for a set of principal paths or new instances for a set of alternate paths. If a group update is selected as a winning update, then the results of the original processing of the update are not altered; therefore, configuration write consistency is maintained in this case. If a group update is selected as a losing update, the resolution algorithm will place the instances created by the update in new alternate paths if the original processing of the update created all principal versions. By placing the set of instances in alternate paths, configuration write consistency is maintained for losing group updates.

Figure VI.18 presents the algorithm for adding to the model all new and existing path and instance records created by updates executed in any of the merging partitions.

Lemma 20: The resolution algorithm for resolving update conflicts adds a mutually consistent set of instance records to the model for each updated object.

Proof of Lemma 20: The resolution algorithm for resolving conflicting updates consists of two parts (a) selecting the winning group updates, and (b) adding the winning and losing update paths to the system directory.

(a) By lemma 17, the set of winning group updates are selected in a mutually consistent manner among the set of merging sites.

(b) Adding the winning and losing update paths to the model involves adding instance records as is from the change list to the model, adding modified fields from path records from the change list to the model, and adding newly created alternate path records to the model. The algorithm for building the model copies instance records as is from the change lists. The change list information is processed in order by path token and, for each path, by the name of the virtual partition reporting an update to the path. This produces a total ordering on the processing. Each of the new instance records are added to the model the first time they are encountered; therefore, each of the merging sites copies the same new instance records from the change lists. The only path record fields modified when resolving update conflicts are the current instance reference and the partition tag associated with updating the path. By lemma 17, these fields values are mutually consistent because they are selected by the algorithm for selecting winning and losing updates. Every merging site creates an identical new alternate path if (i) the path has the same implicit alias number assigned to it; (ii) the path has the same path token; and (iii) the path references the same current instance.

(i) By Lemma 19, the new alternate paths are assigned implicit alias numbers in a mutually consistent manner at every merging site.

(ii) and (iii) These two criteria are equivalent because the new path tokens are formed by adding one to the number portion of the instance token associated with the current instance in the path. The algorithm for adding the losing instance records to the model processes the instances in successor order, that is from oldest to most current. The change lists contain the successor information for each instance and all sites receive the same change lists; therefore, the processing order is well-defined and all merging sites must determine the same instance as the current instance for a single given path.

```
Merging Update Results for an Object O

Calculate A = the largest implicit alias number used in object O
For each updated path from any change list (by increasing path token) Do
        Set No_Winner to TRUE
        Set Winning_Partition to NULL
        Set Last_Non_Type1_Update_Partition to NULL
        For each updating partition (by decreasing partition name) Do
                Consider the oldest instance record reported in this partition's version of this path
                Look up the group update token associated with this instance record
                                in the group update control information
                If the update is a winner
                Then Set No_Winner to FALSE
                        Set Winning_Partition to this partition name
                If this update is NOT processed by group update rule #1
                Then Set Last_Non_Type1_Update_Partition to this partition name
        EndFor
        For each updating partition (by decreasing partition name) Do
                If this partition is the Winning_Partition
                Then Add the path record to the model, if not present
                        While the update is a winner and there is a new instance in
                                        this partition's version of this path Do
                                Add the instance record to the model, if not present
                                Make this instance the current of the model path
                                If there is a successor instance in this version of the path
                                Then Look up the next group update token in the control info
                        EndWhile
                Else If No_Winner AND the partition is the Last_Non_Type1_Update_Partition
                        Then Add the path record to the model, if not present
                                While there is a new successor instance in this partition's version of
                                        this path and the update was not processed by update rule #1 Do
                                  Add the instance record to the model, if not present
                                  Make this instance the current of the model path
                                  If there is a successor instance in this version of the path
                                  Then Look up the next group update token in the control info
                                EndWhile
                        EndIf
                EndIf
                If any updates remain in this partition's version of this path
                Then Make a new path record and add it to the model
                        Increment A and assign the new implicit alias number to this path
                        For each instance record in this partition's version
                                                of this path in successor order Do
                                Add the instance record to the model, if not present
                                Make this instance the current of the new model path
                                Make this instance record reference the new path record as its path
                        EndFor
                        Assign the new path a token = (last instance token's # + 1, sitename)
                EndIf
        EndFor
EndFor
```

Figure VI.18 - Algorithm for Completing the Resolution of Conflicting Updates.

If the instance is the same at every site, its instance token is the same and the path token assigned to the new alternate path record must be the same. Every site creates the same set of new alternate paths.

Therefore, the resolution algorithm for resolving update conflicts adds mutually consistent instance records to the model for each updated object.

Proposition 7: The resolution and propagation performed by merge recovery results in a mutually consistent system directory for the set of merging sites.

Proof of Proposition 7: By lemma 6, the merge algorithm propagates new results in a mutually consistent manner among the merging sites. By lemma 8, lemma 9, lemma 10, lemma 11, lemma 13 , and lemma 20, the merge algorithm produces mutually consistent model records for naming conflicts, assign conflicts, ownership conflicts, permission conflicts, duplicate implicit alias numbers, and update conflicts. By lemma 7, we know that these mutually consistent model records become mutually consistent records in each local system directory by application of the incorporation algorithm. By lemma 16, the (deletion, update) conflicts result in mutually consistent records in each local system directory. Therefore, the resolution algorithm produces a mutually consistent system directory among the set of merging sites.

6.8 Notification of New Alternate Versions

Resolving conflicting updates by creating new alternate versions does not change the user data that is stored by the system, but it does change the name used to reference some of that data. When a client C performs the operation "update Obj1", a successful return means that the version of Obj1 installed by this update is named or can be referenced by the name Obj1. If at a later time a merge recovery concludes that this update conflicts with an update performed in another partition and this update is the losing update, then the new data installed by client C will be placed in an alternate version of Obj1 and must be referenced by the name Obj1(#), where # is the appropriate implicit alias number for this alternate version path. Client C must be notified of this change of name.

The method and form of the notification depends on the client and the environment in which the storage system is implemented. If the client is an end-user, then electronic mail is

a reasonable method for notifying the client of conflicts resolved by a merge recovery. If the client is another software system, a reasonable method for notifying a system is to execute a special purpose program designed for the client system. It would be the responsibility of the client system itself to provide the program to be executed. The storage system would notify such a client by executing the client's specified program.

The notification itself must specify what the conflict was, what client performed the conflicting request, what resolution was performed, and what changes have been made to the mapping of external names to user data instances. Both the winning and the losing client should be informed of the conflict and its resolution. Sending a notification to the winning client is useful if the conflicting clients wish to negotiate a resolution different from the storage system resolution. When the clients are end-users, it is likely that they will talk about the conflicts and negotiate a true merge of their respective modifications. Regardless of which client types are involved in the conflict, the responsibility for acting on a change to the name mapping lies with the clients themselves.

6.9 Atomic Merge Recovery

When requests are processed by the storage system they are performed atomically. The same is true of a merge recovery. Either all results of the recovery are committed or none of the results are committed. Atomicity is at risk when a site or communication failure occurs during merge recovery. When a site fails to complete a merge, the site reverts to the previous state and begins negotiating a new virtual partition from that state.

If the merge recovery is performed in an atomic manner, the previous state of the system may be restored at any time prior to committing all results of the merge recovery. Atomicity of a merge recovery can be accomplished in one of two ways. A site may write and maintain a special undo log that records all modifications made during the merge. If the merge fails due to a site crash or loss of communication with a representative site, the undo log is used to remove all results of the merge process from the local system directory. A second approach is to make a checkpoint of the entire local system directory prior to processing the merge information. The merge recovery is performed on one copy of the

system directory. If the merge process fails, the modified local directory is replaced by the checkpoint version, moving the state of the site back to the state held prior to the attempted merge.

Both approaches are sufficient. Each has advantages and disadvantages. Undo logging may require a large amount of temporary disk space. The log must be implemented as a write-ahead log, meaning that each modification recorded by a log entry must be physically written before the modification it describes is written to the storage device. To ensure that the log is written ahead of the modification, some number of forced writes must be performed. Forced writes are inherently slow as the system must block waiting for the write operation to be completed to the physical device itself. Also, the system implementor must determine exactly what information must be written for each action taken by the merge process and implement the logging process according to this analysis.

Making a checkpoint of the entire system directory requires a large amount of temporary disk space. Creation of the checkpoint can be done by stream reading the local system directory and stream writing a second copy of the directory. Using one forced write ensures that the entire checkpoint copy has been physically written. The main disadvantage of checkpointing is that if a merge has small change lists that result in few modifications to the local system directory, the space and time used to create the checkpoint are substantially larger than the space and time that would be used to maintain a special undo log.

CHAPTER SEVEN
EXAMPLES USING THE UPDATE RESOLUTION ALGORITHM

In this chapter, we present two examples in which we apply the update resolution algorithm in consecutive merge recoveries. The first example illustrates two consecutive merge recoveries for three sites that have performed updates on a single object. The second example illustrates two consecutive merge recoveries for three sites that have performed group updates on a set of objects. The examples are designed to illustrate a number of the most interesting properties of the update resolution algorithm.

7.1 Example One

In our first example we consider three sites A, B, and C and the partitioning behavior shown in figure VII.1. Initially all three sites are members of partition 1A. A communication failure results in the creation of virtual partitions 3A and 3C. A second communication failure between sites A and B result in virtual partitions 5A and 5B. A merger of 5B and 3C produces partition 7C, which in turn merges with partition 5A to form partition 9B. This example illustrates the effects one merge has on a subsequent merge.

An object Obj1 is created in partition 1A. The object Obj1 consists of a single instance I1 and is not updated during the existence of partition 1A. Figure VII.2 shows the updates performed by sites A, B, and C on Obj1 in partitions 3A, 3C, and 5B. These updates constitute the updates of interest for the merge recovery forming partition 7C.

During partition 3A, two updates were executed adding two instances I2 and I3 as successors to instance I1. The partition tag value associated with these three instances is 3A. Partition 3C executed three update to the principal version of Obj1 adding three instances I4, I5, and I6 as successors to instance I1. The partition tag value associated with these three instances is 3C. No updates were performed during partition 5B; therefore, the view of Obj1 in partition 5B is the same of the view of Obj1 held in partition 3A. Partition 5B and 3C merge to form partition 7C. Partition 5B has partition history 1A, 2A, 3A, 4B, and 5B.

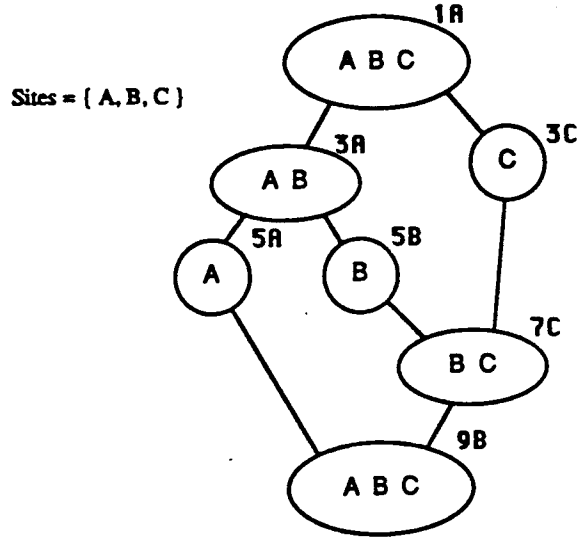


Figure VII.1 - Example One: Partitioning Behavior for Sites A, B, and C.

Updates with an associated partition tag value = 3A:
$G_{3A,1}$ = update Obj1 = { I2 }
$G_{3A,2}$ = update Obj1 = { I3 }

Update with an associated partition tag value = 3C:
$G_{3C,1}$ = update Obj1 = { I4 }
$G_{3C,2}$ = update Obj1 = { I5 }
$G_{3C,3}$ = update Obj1 = { I6 }

Updates with an associated partition tag value = 5B:
None

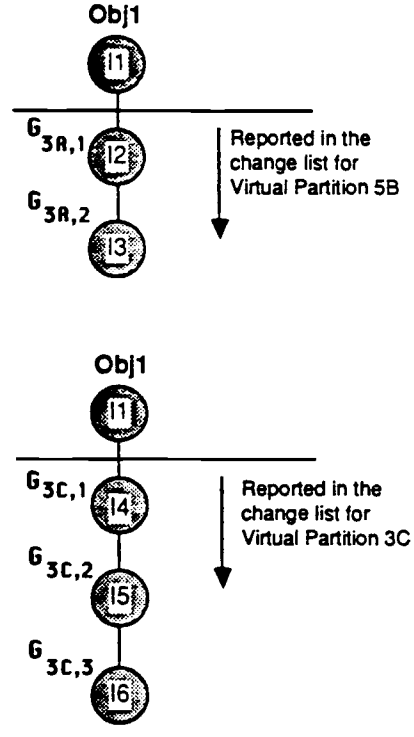


Figure VII.2 - Example One: Updates Performed in Three Partitions.

Partition 3C has partition history 1A, 2C, and 3C. The representative of partition 5B builds a change list containing all information with an associated partition tag value of 2A, 3A, 4B, or 5B. The representative of partition 3C builds a change list containing all information with an associated partition tag value of 2C or 3C. Figure VII.2 illustrates the instances that are included in the two change lists. Information on instance I1 is included in the change list only in the sense that instance I2 and instance I4 refer to it as their immediate predecessor. From the change list control information on group updates we must compute the set of predecessor updates and the set of successor updates for each group update reported prior to selecting the winning updates. Figure VII.3 shows the information computed for merging partitions 5B and 3C. The goodness value assigned to each update, the name of the partition reporting the update, and the group update token assigned to the update are used to prescribe a processing order for selecting the winning updates.

Group Update	Paths Updated	Update Rule	Predec Updates	#Predec Updates	Successor Updates	#Entities Updated	Goodness Value
$G_{3A,1}$	Obj1	1	{ }	0	$\{G_{3A,2}\}$	1	2
$G_{3A,2}$	Obj1	1	$\{G_{3A,1}\}$	1	{ }	1	3
$G_{3C,1}$	Obj1	1	{ }	0	$\{G_{3C,2}, G_{3C,3}\}$	1	2
$G_{3C,2}$	Obj1	1	$\{G_{3C,1}\}$	1	$\{G_{3C,3}\}$	1	3
$G_{3C,3}$	Obj1	1	$\{G_{3C,1}, G_{3C,2}\}$	2	{ }	1	4

Goodness Value:

(2 * #Entities Updated) + #Predecessor Updates

Prescribed Processing Order:

$G_{3C,3}$, $G_{3C,2}$, $G_{3A,2}$, $G_{3C,1}$, $G_{3A,1}$

Winners = { $G_{3C,3}$, $G_{3C,2}$, $G_{3C,1}$ }

Losers = { $G_{3A,2}$, $G_{3A,1}$ }

Figure VII.3 - Example One: Information Required for Update Resolution in Partition 7C.

To select the winning and losing updates we begin by selecting update $G_{3C,3}$ and its predecessors $G_{3C,2}$ and $G_{3C,1}$ as winning updates. The partition tags associated with the instances created by these winning updates retain their current value of 3C. The three winning updates are removed from the processing list. The next update to be considered is $G_{3A,2}$. This update conflicts with all three updates in the set of winning updates; therefore, update $G_{3A,2}$ and all of its successor updates are placed in the set of losing updates. The partition tag associated with the instances created by these losing updates is set to the new partition name 7C. The last update to be considered is update $G_{3A,1}$. This update also conflicts with all three winning updates; therefore, update $G_{3A,1}$ and all of its successor updates are placed in the set of losing updates. The partition tag for these losing updates is set to the new partition name 7C.

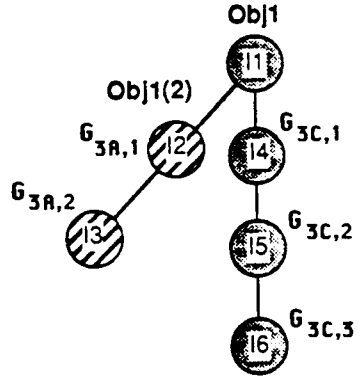


Figure VII.4 - Example One: Results of the Merge Forming Partition 7C.

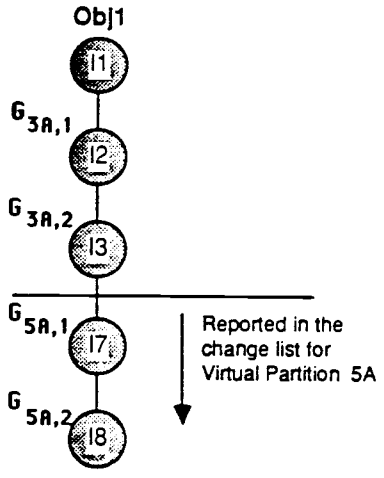
When building the model for object Obj1, the two losing updates will form an alternate version path Obj1(2). Figure VII.4 gives a graphical view of Obj1 following the merge recovery. The clients who created instance I2 and instance I3 will be notified that their updates to the principal path of Obj1 were in conflict with the updates $G_{3C,1}$, $G_{3C,2}$, and $G_{3C,3}$. The set of clients who requested the updates $G_{3C,1}$, $G_{3C,2}$, and $G_{3C,3}$ are notified of their conflict with $G_{3A,1}$ and $G_{3A,2}$. The notification would inform each client of the result of his update, specify the name that should be used to reference the update result, and identify those clients whose updates were in conflict.

Updates with an associated partition tag value = 5A:
$G_{5A,1}$ = update Obj1 = { I7 }
$G_{5A,2}$ = update Obj1 = { I8 }

Updates with an associated partition tag value = 7C:
$G_{7C,1}$ = update Obj1 = { I9 }
$G_{7C,2}$ = update Obj1 = { I10 }
$G_{7C,3}$ = update Obj1 = { I11 }
$G_{7C,4}$ = update Obj1(2) = { I12 }
$G_{3A,1}$ = update Obj1(2) = { I2 }
$G_{3A,2}$ = update Obj1(2) = { I3 }

Updates with an associated partition tag value = 3C:
$G_{3C,1}$ = update Obj1 = { I4 }
$G_{3C,2}$ = update Obj1 = { I5 }
$G_{3C,3}$ = update Obj1 = { I6 }

Updates with an associated partition tag value = 5B:
None



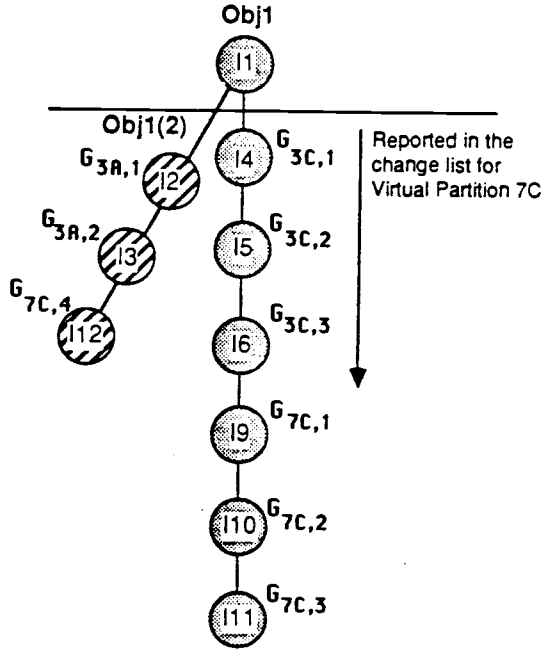


Figure VII.5 - Example One: Updates Performed in Two Partitions.

Partition 7C performs updates on Obj1 and Obj1(2). Partition 5A has no knowledge of the existence of Obj1(2), so partition 5A performs updates on Obj1 only. Figure VII.5 shows the update requests executed in partitions 7C and 5A. The users in partition 7C performed three updates on Obj1, creating instances I9, I10, and I11 as successors of I6, and one update on Obj1(2), creating instance I12 as the successor of I3. These four new instances have associated partition tag values of 7C. The users in partition 5A performed

two updates on Obj1, creating instances I7 and I8 as successors of I3. The instances I7 and I8 have associated partition tag values of 5A. We merge partition 5A and 7C to form partition 9B. Partition 5A has a partition history 1A, 2A, 3A, 4A, and 5A. Partition 7C has a partition history 1A, 2A, 3A, 2C, 3C, 4B, 5B, 6C, and 7C. The representative for partition 5A will construct a change list containing all modifications with an associated partition tag of 4A or 5A. The representative for partition 7C will construct a change list containing all modifications with associated partition tags of 2C, 3C, 4B, 5B, 6C, or 7C.

From the change list information supplied by the two representatives we compute the predecessor updates and the successor updates for each reported group update. Figure VII.6 summarizes the information on all updates reported by the merging partitions. Note that update $G_{5A,1}$, executed in partition 5A, has 2 predecessor updates $G_{3A,2}$ and $G_{3A,1}$. These predecessor updates were reported by partition 7C. Similarly update $G_{5A,2}$ executed in partition 5A has updates $G_{3A,2}$ and $G_{3A,1}$ as predecessors. This occurs because partition 5A is not aware that the instance I3, created by $G_{3A,2}$, and the instance I2, created by $G_{3A,1}$, have been moved to an alternate version path. Update $G_{3A,1}$ and $G_{3A,2}$ are reported by partition 7C as updates to Obj1(2) rather than as updates to Obj1. This occurs as a result of the previous merge resolution.

The set of updates being considered are sorted by their goodness values, the name of the partition reporting the update, and the group update token associated with the update. Update $G_{7C,3}$, which updates the principal path of Obj1, is the first update to be processed. This update and all of its predecessor updates ($G_{7C,2}$, $G_{7C,1}$, $G_{3C,3}$, $G_{3C,2}$, and $G_{3C,1}$) are added to the set of winning updates. The next remaining update to be considered is update $G_{5A,2}$, reported by partition 5A. $G_{5A,2}$ updates the principal path of Obj1. This update conflicts with the principal path updates reported by partition 7C. The updates reported by partition 7C for the principal path of Obj1 have been selected as winning updates; therefore, update $G_{5A,2}$ and all of its successors must be added to the set of losing updates. The update $G_{5A,2}$ has no successors. The partition tag associated with each losing update is set to 9B. The next update considered is update $G_{5A,1}$. This update, reported by partition 5A, is an update to the principal path of Obj1. The update conflicts with updates in the set of winning updates and must be added to the set of losing updates. The next update considered is update $G_{7C,4}$. This is the first update to the alternate path Obj1(2). The update does not conflict with any winning updates. The predecessors $G_{3A,1}$ and $G_{3A,2}$ update Obj1(2).

Group Update	Paths Updated	Update Rule	Predec Updates	#Predec Updates	Successor Updates	#Entities Updated	Goodness Value
$G_{5A,1}$	Obj1	1	$\{G_{3A,2}, G_{3A,1}\}$	2	$\{G_{5A,2}\}$	1	4
$G_{5A,2}$	Obj1	1	$\{G_{5A,1}, G_{3A,2}, G_{3A,1}\}$	3	$\{\ \}$	1	5
$G_{7C,1}$	Obj1	1	$\{G_{3C,3}, G_{3C,2}, G_{3C,1}\}$	3	$\{G_{7C,2}, G_{7C,3}\}$	1	5
$G_{7C,2}$	Obj1	1	$\{G_{7C,1}, G_{3C,3}, G_{3C,2}, G_{3C,1}\}$	4	$\{G_{7C,3}\}$	1	6
$G_{7C,3}$	Obj1	1	$\{G_{7C,2}, G_{7C,1}, G_{3C,3}, G_{3C,2}, G_{3C,1}\}$	5	$\{\ \}$	1	7
$G_{7C,4}$	Obj1(2)	1	$\{G_{3A,1}, G_{3A,2}\}$	2	$\{\ \}$	1	4
$G_{3A,1}$	Obj1(2)	1	$\{\ \}$	0	$\{G_{3A,2}, G_{7C,4}\}$	1	2
$G_{3A,2}$	Obj1(2)	1	$\{G_{3A,1}\}$	1	$\{G_{7C,4}\}$	1	3
$G_{3C,1}$	Obj1	1	$\{\ \}$	0	$\{G_{3C,2}, G_{3C,3}, G_{7C,1}, G_{7C,2}, G_{7C,3}\}$	1	2
$G_{3C,2}$	Obj1	1	$\{G_{3C,1}\}$	1	$\{G_{3C,3}, G_{7C,1}, G_{7C,2}, G_{7C,3}\}$	1	3
$G_{3C,3}$	Obj1	1	$\{G_{3C,1}, G_{3C,2}\}$	2	$\{G_{7C,1}, G_{7C,2}, G_{7C,3}\}$	1	4

Goodness Value:
(2 * #Entities Updated) + #Predecessor Updates

Prescribed Processing Order:
$G_{7C,3}, G_{7C,2}, G_{7C,1}, G_{5A,2}, G_{7C,4}, G_{3C,3}, G_{5A,1}, G_{3C,2}, G_{3A,2}, G_{3C,1}, G_{3A,1}$

Winners = $\{ G_{7C,3}, G_{7C,2}, G_{7C,1}, G_{3C,3}, G_{3C,2}, G_{3C,1}, G_{7C,4}, G_{3A,1}, G_{3A,2} \}$

Losers = $\{ G_{5A,2}, G_{5A,1} \}$

Figure VII.6 - Example One: Information Required for Update Resolution in Partition 9B.

None of these updates conflict with any of the winning updates. All three updates are added to the set of winning updates.

The two losing updates $G_{5A,2}$ and $G_{5A,1}$ created temporally related instances; therefore these instances will be grouped together and placed in one new alternate path Obj1(3). Figure VII.7 presents a graphical view of object Obj1 after the completion of the merge recovery. The clients who requested the losing updates to the principal path of Obj1 are notified of their conflict with the updates $G_{7C,3}$, $G_{7C,2}$, $G_{7C,1}$, $G_{3C,3}$, $G_{3C,2}$, and $G_{3C,1}$. The update $G_{5A,2}$ and $G_{5A,1}$ are in conflict with this set of updates because these are the winning update applied to the principal path of Obj1. The clients who performed the winning updates are also informed of the conflicts.

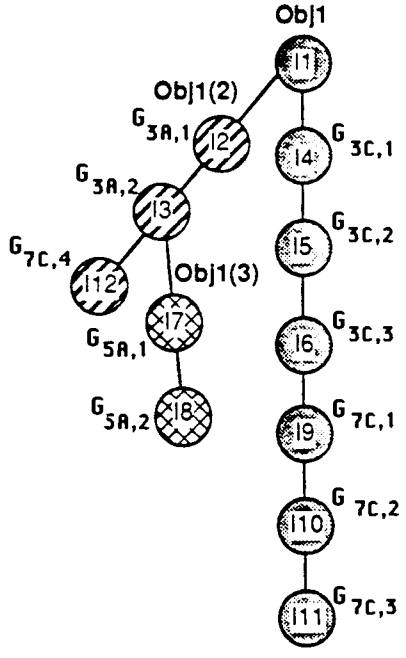


Figure VII.7 - Example One: Results of the Merge Forming Partition 9B.

7.2 Example Two

In our second example we consider the sites A, B, and C, and the same partitioning behavior studied in example one. Initially all three sites are members of partition 1A. In virtual partition 1A three objects Obj1, Obj2, and Obj3 are created. Obj1 is represented by the instance I1.1, Obj2 is represented by the instance I2.1, and Obj3 is represented by the instance I3.1. These objects are not updated in partition 1A. A communication failure results in the creation of virtual partitions 3A and 3C. Several updates are performed on the three objects by users in partitions 3A and 3C. A second communication failure between sites A and B results in virtual partitions 5A and 5B. Additional updates are performed in partitions 5A and 5B. Figure VII.8 shows the update requests executed in partitions 3A, 3C, and 5B.

In partition 3A, two updates are performed. Update $G_{3A,1}$ adds instance I2.2 to the principal path of Obj2 as the successor of instance I2.1. Update $G_{3A,2}$ adds the instance I1.2 to the principal path of Obj1 as a successor of instance I1.1 and the instance I2.3 to the principal path of Obj2 as a successor of I2.2. The updates performed by users in partition 5B are based on the view of Obj1, Obj2, and Obj3 held in partition 3A. The update $G_{5B,1}$ adds instance I2.4 to the principal path of Obj2 as the successor of instance I2.3, which was created by an update in partition 3A. The update $G_{5B,2}$ adds instance I1.4 to the principal path of Obj1 as the successor of instance I1.2, which was created by an update in partition 3A.

In partition 3C, three updates are performed. Update $G_{3C,1}$ adds instance I3.2 to the principal path of Obj3 as the successor of the initial instance I3.1. Update $G_{3C,2}$ adds instance I3.3 to the principal path of Obj3 as the successor of instance I3.2. $G_{3C,3}$ is an update of the principal path of Obj1 and the principal path of Obj3. Instance I1.3 is added to Obj1 as the successor of the initial instance I1.1. Instance I3.4 is added to Obj3 as the successor of instance I3.3.

When partitions 5B and 3C merge to form partition 7C, we begin by comparing the partition history of partition 5B and 3C. The history of partition 5B is 1A, 2A, 3A, 4B, and 5B. The history of partition 3C is 1A, 2C, and 3C. The representative site for partition 5B constructs a change list containing information on all directory entries with an associated partition tag value of 2A, 3A, 4B, or 5B. The representative site for partition 3C constructs a

Updates with an associated partition tag value = 3A:

$G_{3A,1}$ = update Obj2 = { I2.2 }

$G_{3A,2}$ = update Obj1 Obj2 = { I1.2, I2.3 }

Update with an associated partition tag value = 3C:

$G_{3C,1}$ = update Obj3 = { I3.2 }

$G_{3C,2}$ = update Obj3 = { I3.3 }

$G_{3C,3}$ = update Obj1 Obj3 = { I1.3, I3.4 }

Updates with an associated partition tag value = 5B:

$G_{5B,1}$ = update Obj2 = { I2.4 }

$G_{5B,2}$ = update Obj1 = { I1.4 }

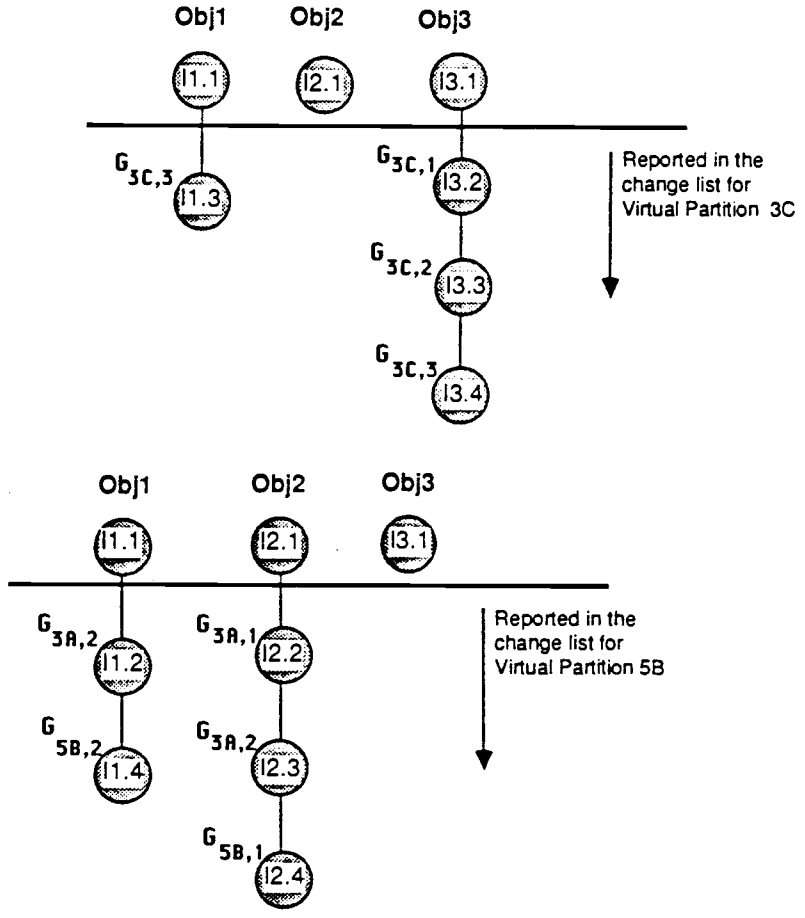


Figure VII.8 - Example Two: Updates Performed in Three Partitions.

change list containing entries with an associated partition tag value of 2C and 3C. Figure VII.8 shows the instances contained in each of the change lists. The original instances I1.1, I2.1, and I3.1 are referenced in the two change lists as the immediate predecessors of several of the new instances. To merge the updates we compute the predecessor updates and the successor updates of each update reported in a change list. Figure VII.9 shows the information calculated by the merge process for use in selecting the winning and losing updates.

Group Update	Paths Updated	Update Rule	Predec Updates	#Predec Updates	Successor Updates	#Entities Updated	Goodness Value
$G_{3A,1}$	Obj2	1	{ }	0	$\{G_{3A,2}, G_{5B,1}\}$	1	2
$G_{3A,2}$	Obj1 Obj2	1	$\{G_{3A,1}\}$	1	$\{G_{5B,1}, G_{5B,2}\}$	2	5
$G_{3C,1}$	Obj3	1	{ }	0	$\{G_{3C,2}, G_{3C,3}\}$	1	2
$G_{3C,2}$	Obj3	1	$\{G_{3C,1}\}$	1	$\{G_{3C,3}\}$	1	3
$G_{3C,3}$	Obj1 Obj3	1	$\{G_{3C,1}, G_{3C,2}\}$	2	{ }	2	6
$G_{5B,1}$	Obj2	1	$\{G_{3A,1}, G_{3A,2}\}$	2	{ }	1	4
$G_{5B,2}$	Obj1	1	$\{G_{3A,2}\}$	1	{ }	1	3

Goodness Value:
(2 * #Entities Updated) + #Predecessor Updates

Prescribed Processing Order:
$G_{3C,3}, G_{3A,2}, G_{5B,1}, G_{5B,2}, G_{3C,2}, G_{3C,1}, G_{3A,1}$

Winners = { $G_{3C,3}, G_{3C,2}, G_{3C,1}, G_{3A,1}$ }
Losers = { $G_{3A,2}, G_{5B,1}, G_{5B,2}$ }

Figure VII.9 - Example Two: Information Required for Update Resolution in Partition 7C.

The group updates are sorted for processing by goodness value, the name of the reporting virtual partition, and the name of the group update. The first update to be processed is $G_{3C,3}$. This update and its predecessor updates $G_{3C,2}$ and $G_{3C,1}$ are added to the set of winning updates. All instances created by these updates retain their original partition tag value of 3C.

The next update to be considered is $G_{3A,2}$, reported by partition 5B. This update conflicts with winning update $G_{3C,3}$ because both requests update Obj1 and the requests were reported by different partitions; therefore this update and its successor updates $G_{5B,1}$ and $G_{5B,2}$ are added to the set of losing updates. The partition tags associated with each instance created by these three updates are set to 7C. The set of clients who performed the update requests $G_{3A,2}$, $G_{5B,1}$, and $G_{5B,2}$ are notified of the conflict with the request $G_{3C,3}$. The client who requested the update $G_{3C,3}$ is notified of its conflict with the other three update requests.

Update $G_{3A,1}$ is the only update remaining to be processed. $G_{3A,1}$ updates Obj2 and does not conflict with any of the winning updates; therefore, $G_{3A,1}$ is added to the set of winning updates. Instance I2.2, created by update $G_{3A,1}$ retains its partition tag value of 3A. Note that a winning update $G_{3A,1}$ is the predecessor of a losing update $G_{3A,2}$.

Figure VII.10 gives a graphical view of Obj1, Obj2, and Obj3 after the merge has been performed. Obj3 maintained its single principal path. Obj1 and Obj2 each added a new alternate path to accommodate the instances created by the losing update $G_{3A,2}$, $G_{5B,1}$, and $G_{5B,2}$. Instance I2.2 created by $G_{3A,1}$ has become the current temporal version on the principal path of Obj2.

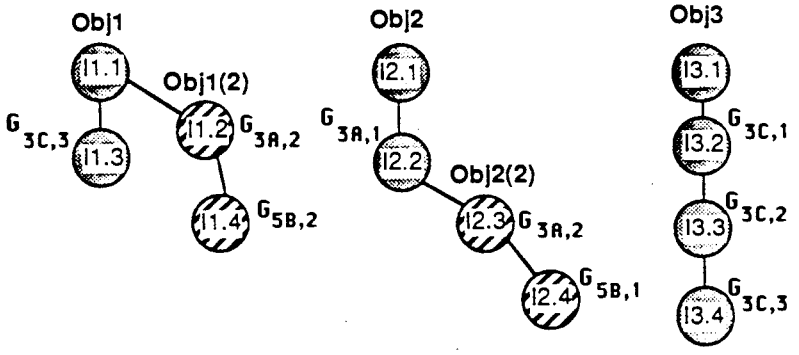


Figure VII.10 - Example Two: Results of the Merge Forming Partition 7C.

The newly formed partition 7C views Obj1, Obj2, and Obj3 as they resulted from the merge process. Updates performed in partition 7C add new instances to this view of the objects. Partition 5A views Obj1, Obj2, and Obj3 as they were at the termination of partition 3A. Updates performed in partition 5A add new instances to the objects based on

Updates with an associated partition tag value = 5A:

$G_{5A,1}$ = update Obj2 = { I2.5 }
$G_{5A,2}$ = update Obj1 = { I1.5 }
$G_{5A,3}$ = update Obj1 = { I1.6 }
$G_{5A,4}$ = update Obj1 Obj2 = { I1.7, I2.6 }

Updates with an associated partition tag value = 7C:

$G_{7C,1}$ = update Obj3 = { I3.5 }
$G_{7C,2}$ = update Obj1 Obj2(2) = { I1.8, I2.7 }
$G_{3A,2}$ = update Obj1(2) Obj2(2) = { I1.2, I2.3 }
$G_{5B,1}$ = update Obj2(2) = { I2.4 }
$G_{5B,2}$ = update Obj1(2) = { I1.4 }

Updates with an associated partition tag value = 3C:

$G_{3C,1}$ = update Obj3 = { I3.2 }
$G_{3C,2}$ = update Obj3 = { I3.3 }
$G_{3C,3}$ = update Obj1 Obj3 = { I1.3, I3.4 }

Updates with an associated partition tag value = 3A:

$G_{3A,1}$ = update Obj2 = { I2.2 }

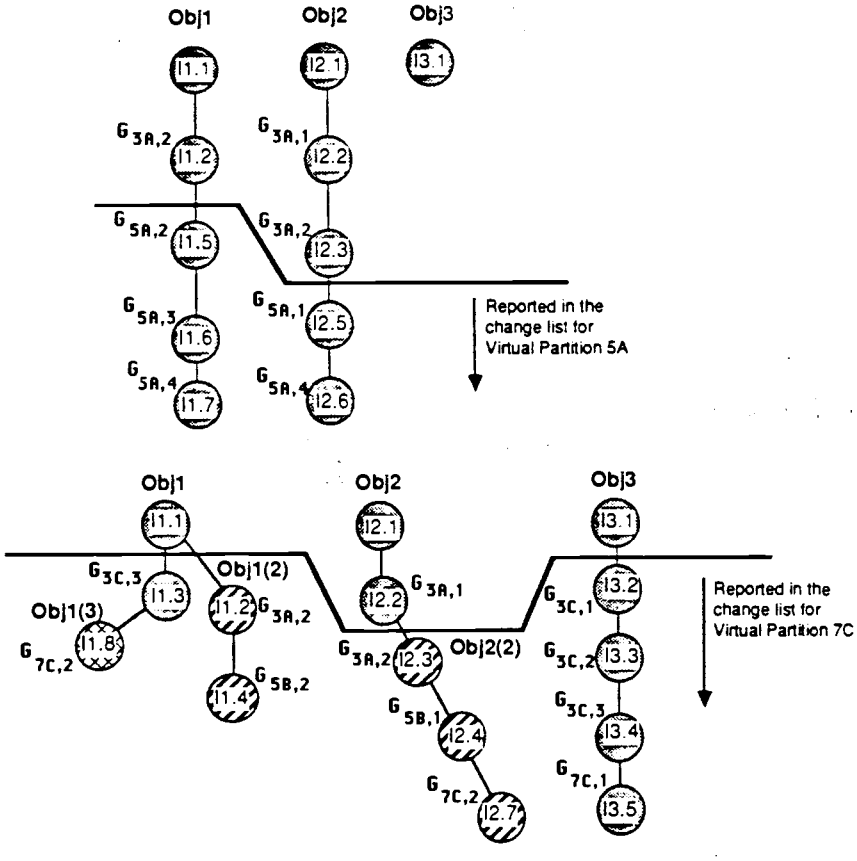


Figure VII.11 - Example Two: Updates Performed in Two Partitions.

this older view. Figure VII.11 presents the updates performed in partition 5A and the updates performed in partition 7C. The effect of these updates on the respective views held by partition 5B and partition 7C are also shown.

In partition 7C, $G_{7C,2}$ is a group update to Obj1 and Obj2(2). The name Obj1 specifies a principal path and the name Obj2(2) specifies an alternate path. This is not an update to a set of principal versions; therefore, group update processing rule four was applied, resulting in the creation of Obj1(3). The update $G_{7C,1}$ adds instance I3.5 to Obj3 as the successor of I3.4.

In partition 5A, four updates are performed. Update $G_{5A,1}$ adds instance I2.5 to Obj2 as the successor of instance I2.3. (The instance I2.3 was the current version of Obj2 at the termination of partition 3A.) Update $G_{5A,2}$ and $G_{5A,3}$ add consecutive instances I1.5 and I1.6 to Obj1. Instance I1.5 is the successor of instance I1.2 (the current version of Obj1 at the termination of partition 3A). $G_{5A,4}$ is a group update on the principal path of Obj1 and the principal path of Obj2.

Partition 9B is formed by merging partition 5A and 7C. The partition history of partition 5A is 1A, 2A, 3A, 4A, and 5A. The partition history of partition 7C is 1A, 2A, 3A, 2C, 3C, 4B, 5B, 6C, and 7C. The representative for partition 5A adds to its change list information on all instances with an associated tag value of 4A or 5A. The representative for partition 7C adds to its change list information on all instances with an associated partition tag value of 2C, 3C, 4B, 5B, 6C, or 7C. Figure VII.11 shows the set of instances that have associated partition tag values in this range. Note that instance I2.2 created by update $G_{3A,1}$ is not included in the change list constructed by either representative site. This update was performed in partition 3A and its status was not altered by the previous merge recovery; therefore, the instance I2.2 retained its partition tag value of 3A. Partition 3A is not in the range of interest for this merge because both of the merging partitions claim knowledge of the results of partition 3A.

Each merging site processes the change list control information to determine the predecessor updates and the successor updates for each reported update. Figure VII.12 shows the information calculated from the two change lists. The update $G_{5B,1}$ appears as an update to Obj2(2) rather than Obj2. $G_{5B,1}$ was a losing update in the previous merge and the instance created by this update is now a member of alternate version Obj2(2). Similarly, the

update $G_{5B,2}$ appears as an update to Obj1(2) and the group update $G_{3A,2}$ appears as an update to Obj1(2) and Obj2(2) because they were both losing updates in the previous merge.

Group Update	Paths Updated	Update Rule	Predec Updates	#Predec Updates	Successor Updates	#Entities Updated	Goodness Value
$G_{5A,1}$	Obj2	1	$\{G_{3A,2}\}$	1	$\{G_{5A,4}\}$	1	3
$G_{5A,2}$	Obj1	1	$\{G_{3A,2}\}$	1	$\{G_{5A,3}, G_{5A,4}\}$	1	3
$G_{5A,3}$	Obj1	1	$\{G_{5A,2}, G_{3A,2}\}$	2	$\{G_{5A,4}\}$	1	4
$G_{5A,4}$	Obj1 Obj2	1	$\{G_{5A,3}, G_{5A,2}, G_{5A,1}, G_{3A,2}\}$	4	$\{\}$	2	8
$G_{7C,1}$	Obj3	1	$\{G_{3C,3}, G_{3C,2}, G_{3C,1}\}$	3	$\{\}$	1	5
$G_{7C,2}$	Obj1(3) Obj2(2)	4	$\{G_{5B,1}, G_{3A,2}, G_{3C,3}\}$	3	$\{\}$	2	7
$G_{3A,2}$	Obj1(2) Obj2(2)	1	$\{\}$	0	$\{G_{5B,2}, G_{5B,1}, G_{7C,2}\}$	2	4
$G_{3C,1}$	Obj3	1	$\{\}$	0	$\{G_{3C,2}, G_{3C,3}, G_{7C,1}\}$	1	2
$G_{3C,2}$	Obj3	1	$\{G_{3C,1}\}$	1	$\{G_{3C,3}, G_{7C,1}\}$	1	3
$G_{3C,3}$	Obj1 Obj3	1	$\{G_{3C,1}, G_{3C,2}\}$	2	$\{G_{7C,1}\}$	2	6
$G_{5B,1}$	Obj2(2)	1	$\{G_{3A,2}\}$	1	$\{G_{7C,2}\}$	1	3
$G_{5B,2}$	Obj1(2)	1	$\{G_{3A,2}\}$	1	$\{\}$	1	3

Goodness Value:
(2 * #Entities Updated) + #Predecessor Updates

Prescribed Processing Order:
$G_{5A,4}, G_{7C,2}, G_{3C,3}, G_{7C,1}, G_{3A,2}, G_{5A,3}, G_{5B,2}, G_{5B,1}, G_{3C,2}, G_{5A,2}, G_{5A,1}, G_{3C,1}$

Winners = $\{ G_{5A,4}, G_{5A,3}, G_{5A,2}, G_{5A,1}, G_{3A,2}, G_{5B,2}, G_{5B,1}, G_{3C,2}, G_{3C,1} \}$
Losers = $\{ G_{7C,2}, G_{3C,3}, G_{7C,1} \}$

Figure VII.12 - Example Two: Information Required for Update Resolution in Partition 9B.

The set of predecessor updates for update $G_{5A,1}$ contains the update $G_{3A,2}$. Update $G_{5A,1}$, reported by partition 5A, claims that instance I2.3 is the predecessor of the instance I2.5. Partition 5A does not report the update that created instance I2.3, but partition 7C does report the update that created instance I2.3. Instance I2.3 is reported because that instance

was placed in the alternate path Obj2(2) during the formation of partition 7C. Instance I2.3 reports instance I2.2 as its predecessor. Neither of the partitions report an update creating instance I2.2; therefore, the chain of predecessors for update $G_{5A,1}$ is complete. Update $G_{5A,2}$ claims that instance I1.2 is the predecessor of the instance I1.5. Partition 5A does not report the update that created instance I1.2, but partition 7C does report the update that created instance I1.2. Instance I1.2 is reported because that instance was placed in the alternate path Obj1(2) as a result of the previous merge recovery. Instance I1.2 reports instance I1.1 as its predecessor. Neither of the partitions report an update creating instance I1.1; therefore, the chain of predecessors for update $G_{5A,2}$ is complete. Calculating the predecessors of update $G_{5A,3}$ is analogous to the calculation we just performed for the update $G_{5A,2}$.

Update $G_{5A,4}$ updates two objects. Calculating the predecessor updates for $G_{5A,4}$ requires that we union the predecessors of the two updated objects. Instance I2.6 and instance I1.7 are created by update $G_{5A,4}$. Update $G_{5A,4}$ reports instance I2.5 as the predecessor of instance I2.6. The change list of partition 5A reports instance I2.5 with a predecessor of I2.3. The instance I2.3 is reported by partition 7C. The updates $G_{5A,1}$ and $G_{3A,2}$ created the instances I2.5 and I2.3 and are members of the set of predecessor updates of $G_{5A,4}$. The second instance created by update $G_{5A,4}$ is I1.7. The instance I1.7 reports instance I1.6 as its predecessor. Instance I1.6 reports instance I1.5 as its predecessor. Update $G_{5A,3}$ created instance I1.6 and update $G_{5A,2}$ created instance I1.5. Both of these updates are reported in the change list of partition 5A and are members of the set of predecessor updates of $G_{5A,4}$. Instance I1.5 reports instance I1.2 as its predecessor. Update $G_{3A,2}$ created instance I1.2 and is reported in the change list of partition 7C; therefore, update $G_{3A,2}$ is a member of the set of predecessor updates of $G_{5A,4}$. The predecessor of instance I1.2 is instance I1.1. The update that created instance I1.1 is not reported in either change list; therefore, we have calculated all of the predecessor updates for update $G_{5A,4}$. The predecessor updates for update $G_{7C,1}$ and update $G_{7C,2}$ are calculated in a similar manner.

Calculation of the successor updates for each update is a simple computation. The successor updates for update $G_{3C,3}$ are interesting in that the update $G_{7C,2}$ is not considered to be a successor of $G_{3C,3}$. The update $G_{3C,3}$ created instance I1.3 which claims membership in the principal path of Obj1. The instance I1.8 is not considered to be a successor of the instance I1.3 because the reporting of instance I1.8 claims membership in

the alternate version path Obj1(3). All successors are members of the same version path and are reported by the same partition.

Based on the calculated predecessor sets and the number of entities updated, we can compute a goodness value for each reported update. The set of group updates are sorted by goodness value, the name of the reporting virtual partition, and the name of the group update. Update $G_{5A,4}$ is the first to be processed. This update and its predecessor updates ($G_{5A,3}$, $G_{5A,2}$, $G_{5A,1}$, and $G_{3A,2}$) are added to the set of winning updates.

The next update to be considered is $G_{7C,2}$. This update to Obj1(3) and Obj2(2) does not conflict with any of the winning updates. We must also determine that none of the predecessor updates of $G_{7C,2}$ are in conflict with any of the winning updates. The updates $G_{5B,1}$, $G_{3A,2}$, and $G_{3C,3}$ are predecessors to $G_{7C,2}$. The update $G_{3A,2}$ is already a member of the set of winning updates. $G_{5B,1}$ updates only Obj2(2) and does not conflict with any update in the set of winning updates. The update $G_{3C,3}$ is a group update to the principal path of Obj1 and the principal path of Obj3. This update conflicts with the updates $G_{5A,4}$, $G_{5A,3}$, $G_{5A,2}$, $G_{5A,1}$, and $G_{3A,2}$ on the principal path of Obj1. Because one of the predecessors of the update $G_{7C,2}$ conflicts with the set of winning updates, the update $G_{7C,2}$ and all of its successors are added to the losing set. $G_{7C,2}$ has no successors, so $G_{7C,2}$ is added to the set of losing updates. The status of the predecessors of $G_{7C,2}$ is not determined on the basis of this failure. The client who requested the update $G_{7C,2}$ is notified of the conflict with the set of updates $G_{5A,4}$, $G_{5A,3}$, $G_{5A,2}$, $G_{5A,1}$, and $G_{3A,2}$. The set of clients who requested the winning updates are also notified of their conflict with the update $G_{7C,2}$.

The next update in the prescribed order is $G_{3C,3}$, reported by partition 7C. This is a group update to the principal paths of Obj1 and Obj3. A winning update has already been selected for the principal path of Obj1 and that update was reported by partition 5A; therefore, the update must be in conflict with the winning updates. The update $G_{3C,3}$ and its successor update $G_{7C,1}$ are added to the set of losing updates. The clients who requested the updates $G_{3C,3}$ and $G_{7C,1}$ are notified of their conflict with the winning updates $G_{5A,2}$, $G_{5A,3}$, and $G_{5A,4}$. The clients who requested the winning updates are also notified of the conflict.

The next unprocessed update in the prescribed order is $G_{5B,2}$. $G_{5B,2}$ is reported by partition 7C as an update to the alternate version path Obj1(2). The update $G_{3A,2}$ is a member of the set of winning updates and $G_{3A,2}$ updates the alternate version path Obj1(2). The updates $G_{5B,2}$ and $G_{3A,2}$ are not in conflict because they are both reported by partition

7C; therefore, they must be consistent updates. The update $G_{3A,2}$ is a predecessor to $G_{5B,2}$. $G_{3A,2}$ is already a member of the set of winning updates; therefore, $G_{3A,2}$ does not conflict with any winning updates. $G_{5B,2}$ is added to the set of winning updates. The next update $G_{5B,1}$ has an analogous relationship to the update $G_{3A,2}$ on the alternate version path Obj2(2). $G_{5B,1}$ is added to the set of winning updates.

The update $G_{3C,2}$ is the next to be processed. $G_{3C,2}$ is reported by partition 7C as an update to the principal path of Obj3. No winning updates have been selected for the principal path of Obj3; therefore, $G_{3C,2}$ does not conflict with any winning updates. The update $G_{3C,1}$ is a predecessor update of $G_{3C,2}$. $G_{3C,1}$ updates only the principal path of Obj3 and has no predecessors. The update $G_{3C,1}$ does not conflict with any of the winning updates. $G_{3C,2}$ and $G_{3C,1}$ are added to the set of winning updates.

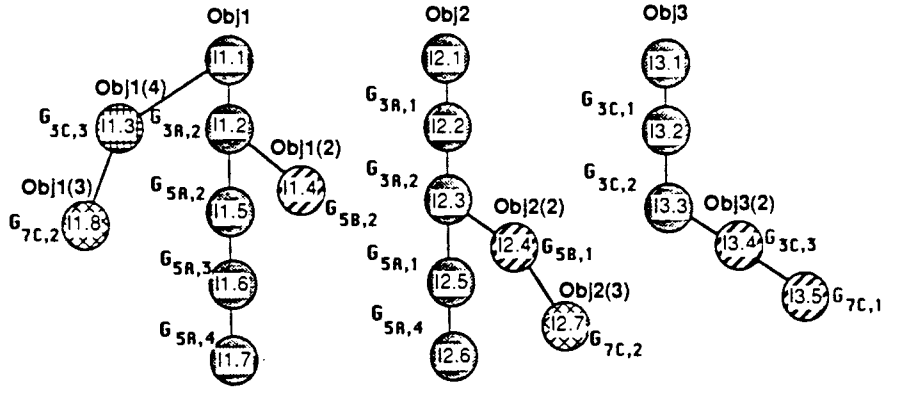


Figure VII.13 - Example Two: Results of the Merge Forming Partition 9B.

All of the reported updates have been processed. Three updates were selected as losing updates: $G_{7C,2}$, $G_{3C,3}$, and $G_{7C,1}$. Figure VII.13 presents a graphical view of Obj1, Obj2, and Obj3 after the merge procedure has completed. The losing updates result in the creation of some alternate version paths. The update $G_{7C,2}$ is a group update which created the instance I1.8 in Obj1(3) and the instance I2.7 in Obj2(2). No other updates were reported for Obj1(3) and $G_{7C,2}$ was not processed under the type one group update rule; therefore $G_{7C,2}$ is selected as a partially successful update on Obj1(3). This means that the instance I1.8 remains in the alternate version path Obj1(3); no new version path is created for the instance I1.8. The current version of Obj1(3) is the instance I1.8. For the version path Obj2(2), update $G_{3A,2}$ was selected as a winning update; therefore, the instance I2.7 is placed in a

new alternate version path Obj2(3). (The implicit alias number 3 is the next available alias number for Obj2.) The current version of Obj2(3) is the instance I2.7.

The losing update $G_{3C,3}$ attempted to update the principal path of Obj1 and Obj3. Winning updates were selected for both of these paths; therefore, $G_{3C,3}$ cannot be selected as a partially successful update. The new alternate version path Obj1(4) is created for the instance I1.3 and the new alternate version path Obj3(2) is created for the instance I3.4. (The implicit alias number 4 is the next available alias number for Obj1 and the implicit alias number 2 is the next available alias number for Obj3.) The current version of Obj1(4) is the instance I1.3.

The last losing update $G_{7C,1}$ attempted to add the instance I3.5 to the principal path of Obj3. A winning update was selected for the principal path of Obj3; therefore, $G_{7C,1}$ cannot be selected as a partially successful update. The instance I3.5 is a successor to the instance I3.4 created by the losing update $G_{3C,3}$. The new alternate version path Obj3(2) has already been created for the instance I3.4. The losing instance I3.5 is added to this same alternate version path Obj3(2). The current version of Obj3(2) is the instance I3.5.

Comparing the result of the first merge with the result of the second merge we see that a resolution selected by the first merge can be undone by the second merge. If we look at the overall result produced by the second merge, we see that only seven instances belong to alternate version paths. The instance I1.8 was specifically placed in an alternate version by the user request $G_{7C,2}$. We may "charge" the merge resolution with creating the other six alternate instances. If the second merge had not undone some alternate versions created by the first merge, the second merge could have resulted in at least ten alternate instances. This example illustrates how the merge resolution algorithm is effective in minimizing the number of losing updates.

CHAPTER EIGHT
EXPERIMENTATION WITH DHSS

In chapter 3, we presented the storage and processing model of DHSS. The results contained in chapter 4, chapter 5, and chapter 6 show that DHSS maintains mutual consistency as a multi-reader/multi-writer system. In this chapter we propose an experiment to explore whether or not a multi-reader/multi-writer system is an improvement over a traditional single-reader/single-write system. One software system S_1 is an improvement over another software system S_2 if S_1 enhances the productivity of its users as compared to the productivity achieved using S_2. Such productivity increases are a function of both the system itself and how it is utilized by its users.

8.1 The Problem

The problem to be investigated is: Is there a difference in the productivity of users of a multi-reader/multi-writer design storage system and the productivity of users of a traditional single-reader/single-writer design storage system? User productivity may be measured both quantitatively and qualitatively. In this experiment, we propose to focus on the quantity of time lost by users due to the functioning of the system. This lost time may be time a user spends waiting to access data that is locked by another user or time spent reviewing and resolving the results of conflict resolutions performed by the system. The independent variable we propose to measure is lost time. Our primary null hypothesis is:

H_0 There is no difference in the lost time experienced by users of a multi-reader/multi-writer design storage system and the lost time experienced by users of a traditional single-reader/single-writer design storage system.

The corresponding alternative hypothesis is:

H_1 The lost time experienced by users of a multi-reader/multi-writer design storage system is less than the lost time experienced by users of a traditional single-reader/single-writer design storage system.

The environment required to carry out an experiment is a team of engineers working on a design project. The designs created and shared by the team members are stored in an experimental distributed storage system. Figure VIII.1 shows the engineers' view of the distributed storage system. The engineers interact with the system by submitting requests and receiving replies. All of the DHSS request types, such as checkout, update, set_permission, and delete are processed by the experimental system. When a user submits a request to the system, the user is blocked from further action until a reply has been received. The experimental system is really two systems: a single-reader/single-writer (SR/SW) system and a multi-reader/multi-writer (MR/MW) system. When an engineer submits a request to the storage system, that request is assigned to execute under one of the two access mechanisms. Users are not informed of which system has executed a particular request, though they may speculate, on the basis of the system response, which system has processed their request.

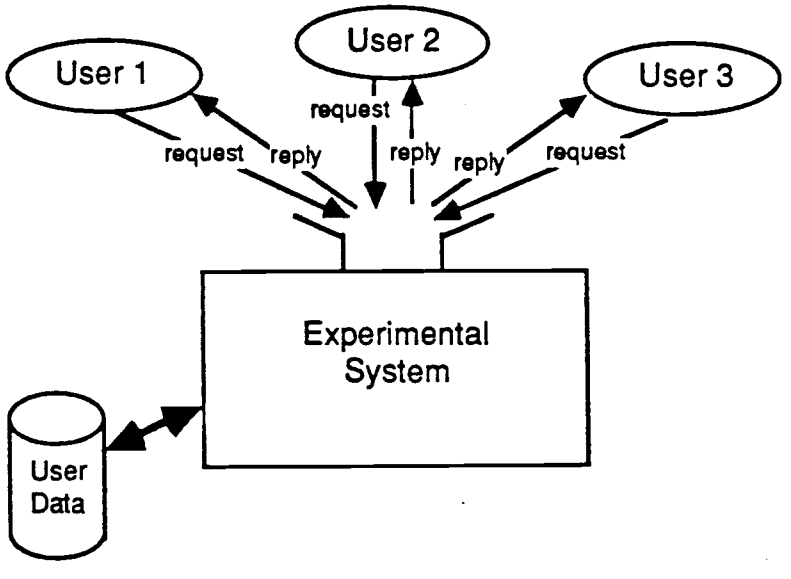


Figure VIII.1 - User's View of the Experimental System.

It is desirable to perform our experiment in an environment where one system is able to behave like either of the two systems. The alternative is to use two systems, one running the SR/SW mechanism and the other running the MR/MW mechanism. This results in a simplier but less reliable experiment. With two separate systems we could have one team use both systems or two different teams each using only one system. One team using both systems would require that the team members repeat their work on each system. This would produce biased results for the second system because each user experiences learning transfer from his work with the first system. Using two teams, one for each system, is equally biased in that we cannot control the difference between the general behavior of the two teams and the difference between the projects these teams are working on. For these reasons we use one system and one team of engineers for our experiment.

In our single system environment, we propose to measure the lost time for each user, for each request performed, under each of the access control mechanisms. These measurements form the set of sample points for the experiment. The DHSS request types are clustered into six classes: create, (checkout + update), derive, assign, (set_permission + grant + deny), and (new_owner + usurp). By organizing our sample data per user per request class we are able to investigate a collection of sub-hypotheses. Let U_i be a user, let R_j be a request class, and let M_k be one of the two access control mechanisms. We define the first set of null sub-hypotheses as follows:

$H_{0_{U_i,R_j}}$ There is no difference in the lost time experienced by a user U_i submitting a request from the class R_j under a multi-reader/multi-writer design storage system and the lost time experienced by the user U_i submitting a request from the class R_j under a traditional single-reader/single-writer design storage system.

The corresponding set of alternative sub-hypotheses is:

$H_{1_{U_i,R_j}}$ The lost time experienced by a user U_i submitting a request from the class R_j under a multi-reader/multi-writer design storage system is less than the lost time experienced by the user U_i submitting a request from the class R_j under a traditional single-reader/single-writer design storage system.

We define the second set of null sub-hypotheses as follows:

$H_{0_{U_i}}$ There is no difference in the lost time experienced by a user U_i under a multi-reader/multi-writer design storage system and the lost time experienced by the user U_i under a traditional single-reader/single-writer design storage system.

The corresponding set of alternative sub-hypotheses is:

$H_{1_{U_i}}$ The lost time experienced by a user U_i under a multi-reader/multi-writer design storage system is less than the lost time experienced by the user U_i under a traditional single-reader/single-writer design storage system.

We define the third set of null sub-hypotheses as follows:

$H_{0_{R_j}}$ There is no difference in the lost time experienced by the team of users submitting requests from the class R_j under a multi-reader/multi-writer design storage system and the lost time experienced by the team of users submitting requests from the class R_j under a traditional single-reader/single-writer design storage system.

The corresponding set of alternative sub-hypotheses is:

$H_{1_{R_j}}$ The lost time experienced by the team of users submitting requests from the class R_j under a multi-reader/multi-writer design storage system is less than the lost time experienced by the team of users submitting requests from the class R_j under a traditional single-reader/single-writer design storage system.

Grouping all users and all requests for each of the mechanisms yields the primary null hypothesis stated earlier.

The sample points in this experiment are characterized by the following model.

$$Y_{m(i,j,k)} = \mu + u_i + r_j + m_k + ur_{j(i)} + um_{k(i)} + rm_{jk} + urm_{jk(i)} + \varepsilon_{m(i,j,k)}$$

$Y_{m(i,j,k)}$ is the lost time observed at the m-th observation; μ is the average lost time over an infinite population; u_i is the random effect due to different users; r_j is the fixed effect due to different request classes; m_k is the random effect of the randomly selected access control mechanism; $ur_{j(i)}$ is the interaction of users and requests; $um_{k(i)}$ is the interaction of users and access control mechanisms; rm_{jk} is the interaction of requests and access control mechanisms; $urm_{jk(i)}$ is the interaction of users, requests, and access control mechanisms; and $\varepsilon_{m(i,j,k)}$ is the random error present in the m-th observation of the i-th user performing a

request from the j-th request class under the k-th mechanism. Before commenting on the expected effect due to users, request classes, and the two mechanisms, we define precisely how the lost time will be measured under each access control mechanism.

8.1.1 Measuring Lost Time in a Single-Reader/Single-Writer System

This experiment is limited to single-reader/single-writer mechanisms that are based on locking. We divide the set of system requests into two logical types: requests that operate on user data and requests that operate on the directory information about user data. For a data item X we will associate one lock with X and one lock with the directory information about X. If a request R attempts to access a data item X, R is blocked from locking X if X is known to be locked by some other request or if, due to a failure, R cannot obtain a lock on X. Similarly, if a request R attempts to access directory information about a data item X, R is blocked from locking the directory information about X if that directory information is known to be locked by some other request or if, due to a failure, R cannot obtain a lock on the directory information for X. All blocked requests are queued in the order of their arrival until the desired lock can be obtained. Figure VIII.2 illustrates the behavior of the SR/SW mechanism when processing a checkout request for a data item that is currently locked by another user. It is clear that the lost time for the user who submitted the checkout request is the time spent in the blocked state waiting to obtain the lock. We define the cost of a request executed under the single-reader/single-writer mechanism to be the time that that request was blocked to obtain a lock. We assume that the time to perform an unblocked request is equal under the two mechanisms; therefore, this time is not a factor in the lost time assigned to the requests processed under either mechanism.

The SR/SW system will maintain the single-reader/single-writer requirement during failures. This implies that the SR/SW mechanism must detect failures and use the knowledge of which sites are running as part of its processing control mechanism. The single-reader/single-writer requirement could be maintained by blocking all requests when a failure condition exists, or by employing a concurrency control mechanism such as fixed primary site or quorums which would allow processing in at most one partition. Any of these approaches would achieve the single-reader/single-writer requirement.

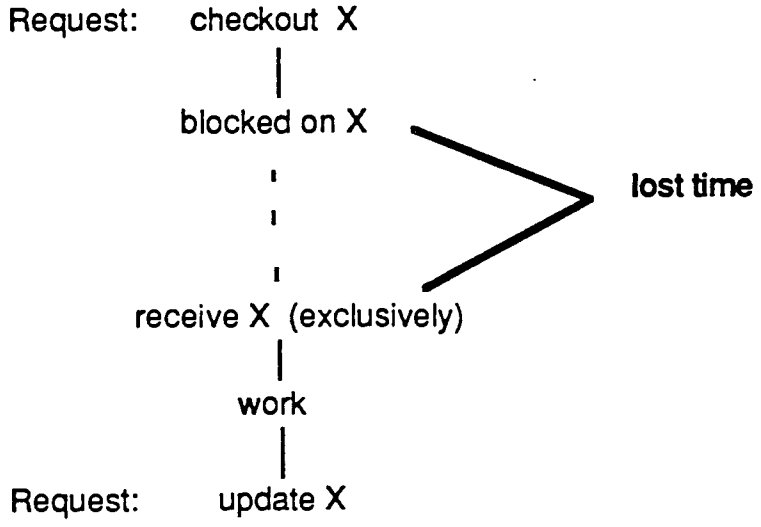


Figure VIII.2 - Processing Behavior of the Single-Reader/Single-Writer System.

8.1.2 Measuring Lost Time in a Multi-Reader/Multi-Writer System

DHSS as described in chapters 3, 4, and 5 will serve as the processing model for the multi-reader/multi-writer mechanism. We divide the set of system requests into two logical types: requests that operate on user data and requests that operate on the directory information about user data. The MR/MW system treats these two types of requests differently. For requests that operate on user data (checkout and update), the MR/MW system does not block the processing of a request in order to obtain a lock. These requests are assigned an initial lost time value of zero. For requests that operate on the directory information about user data, the MR/MW system obtains a lock among the sites in a virtual partition. These requests may be blocked for brief periods of time waiting to obtain a lock. The lost time for such requests is the time a request is blocked waiting to obtain a lock.

Given that the MR/MW system does not lock user data and that directory locks are held among mutually exclusive subsets of sites, conflicts may occur. The MR/MW system resolves these conflicts by creating alternate versions of a design or by selecting one directory modification to prevail over another modification. Users may believe that some of the resolutions performed by the system are undesirable. If a resolution result is undesirable,

the users must compensate by performing additional design work and making additional requests of the system. Figure VIII.3 shows the processing behavior of the multi-reader/multi-writer system for a conflicting checkout and update request. In this case an undesired alternate version was created by the system. To compensate for this undesired behavior, users must examine the conflicting design versions, re-evaluate the modifications made, and coalesce the set of modifications to form one consistent update. The time spent compensating for undesirable results will be divided equally and added to the lost time of the requests that caused the conflicts to occur.

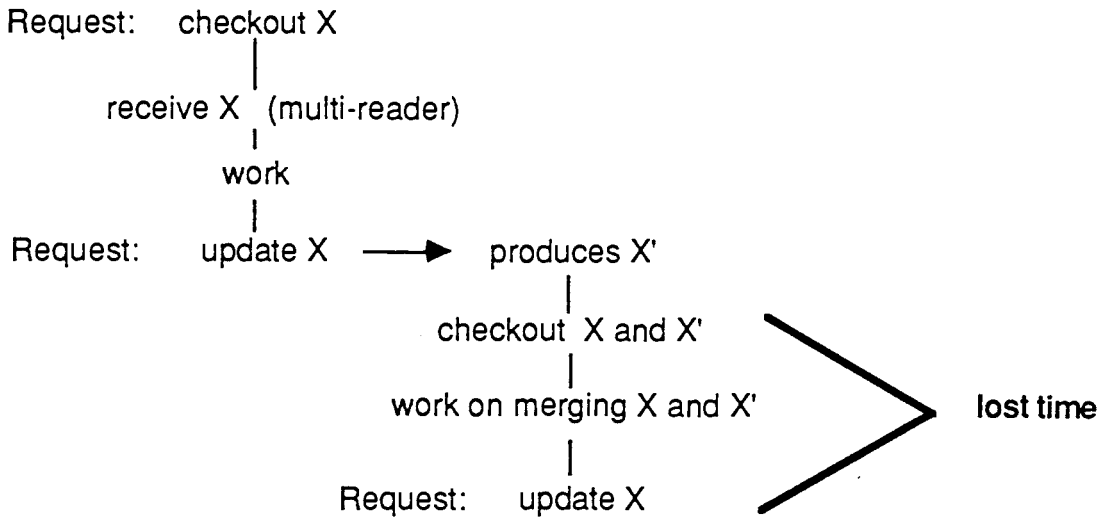


Figure VIII.3 - Processing Behavior of the Multi-Reader/Multi-Writer System.

The requests performed as part of compensation are not sample points for the experiment. In order to exclude these requests from the sample and to identify the conflict-causing requests we have defined a redo request. The redo request has the following form

REDO failure_id new_command

When a conflict occurs, the user receives a notification of that conflict. The notification message contains a failure_id. This failure_id is used by the user in every redo command that is required to compensate for the conflict reported in the notification. When a redo command is processed, the failure_id indicates which request(s) should have additional time added to their associated lost time. We will refer to these requests as the indicated requests. If the new_command is a request that operates on the directory, the time to perform that

operation is divided equally and added to the lost time associated with the indicated requests. If the new_command is a checkout request, a timer is initialized to zero and associated with the failure_id. If the new_command is an update request, the value of the timer associated with the failure_id is divided equally and added to the lost time associated with the indicated requests.

The MR/MW mechanism supports multi-reader/multi-writer access during failures. Both mechanisms must detect failures and the correction of failures. We assume that the time spent detecting failures and detecting the correction of failures is equal in the two systems; therefore, this time is not a factor in the lost time assigned to the requests processed by either system.

8.2 Designing a Random Experiment

In the preceding sections we defined how we would measure the lost time associated with a request performed under the control of either mechanism. The next step is to design a random experiment that will gather these measurements by making independent observations of the SR/SW mechanism and the MR/MW mechanism. The observations should be independent or the results of the experiment could be biased in favor of one mechanism. By independent observation we mean that the behavior of one mechanism will not affect the lost time measurement ascribed to the other mechanism.

Achieving totally independent observations may not be possible in our single system environment. Independence is difficult to guarantee because all requests are carried out on a single set of user data and two contrary mechanisms must be forced to maintain a single consistent view of this data. We will present three possible designs for an experiment. The first design permits independent observation but may not allow us to gather an adequate number of sample points for one of the mechanisms. The second design attempts to overcome the potential shortcomings of the first design by randomizing the experiment on a finer granularity to equalize the number of sample points for each mechanism. The third design allows us to gather approximately equal sample sizes but suffers from some small interaction between the two methods being measured. The degree of the interaction is a function of how the users interact with the system.

All three designs are based on the architecture illustrated in figure VIII.4. When a request enters the system, it is assigned by a switching algorithm to be executed under the SR/SW mechanism or under the MR/MW mechanism. The request is executed under the selected mechanism and a cost is assigned to the request. The assignment of costs is performed as defined for each mechanism. Once execution of the request is completed, a reply is returned to the requesting user. A distinct switching algorithm is defined for each experiment design.

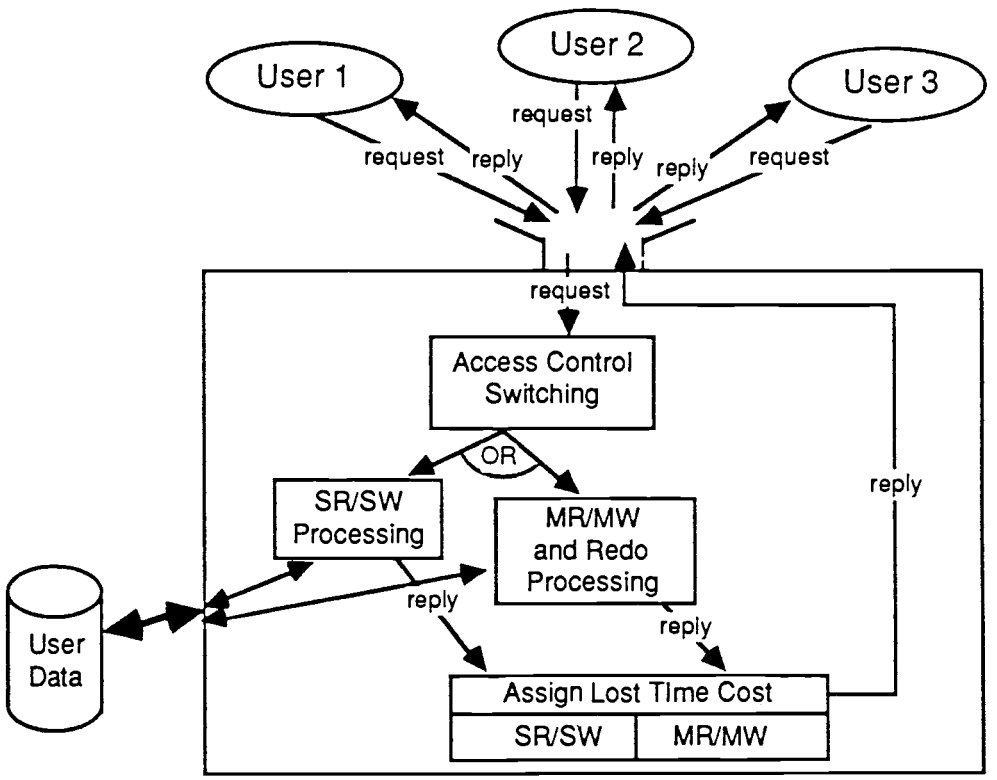


Figure VIII.4 - Detailed View of the Experimental System.

8.2.1 Design of an Experiment: Option 1

In our first design we achieve independent observations by executing all requests under one mechanism for a period of time then switching to the other mechanism for some period of time. Because only one access control mechanism is being employeed at any given time, the system behavior exhibited by one mechanism cannot interfere with the system behavior of the other mechanism. The switching algorithm for our first design is shown in figure VIII.5. When the currently selected access control system is quiescent, we randomly switch from the current mechanism to an alternate mechanism. The test to determine whether or not the system is quiescent depends on the access control method currently in use. If the SR/SW mechanism was being used, the system is deemed to be quiescent if there are no requests being executed and there is no user data checked out. If the MR/MW mechanism was being used, the system is deemed to be quiescent if there are no requests being executed, there is no user data checked out, and there are no outstanding compensating requests to be performed.

Algorithm 1 for Access Control Switching

```
Set current method = random selection of { SR/SW, MR/MW }
While the experiment is in progress Do
        Wait for a user to submit a request
        When a request is received Do
                Case of current method
                        SR/SW:
                                If no failure exists and no locks are held and no requests
                                                                are currently being executed
                                Then Set current method = random selection of { SR/SW, MR/MW }
                        MR/MW:
                                If no failure exists and no data is checked out and no requests
                                                are currently being executed and
                                                there are no outstanding Redo requests
                                Then Set current method = random selection of { SR/SW, MR/MW }
                EndCase
                Process the request under the current method
        EndWhen
EndWhile
```

Figure VIII.5 - Algorithm 1 for Random Selection of an Access Control Mechanism.

The switching algorithm cannot determine whether or not there are any outstanding compensating requests without the assistance of the users. The users assist the system by

using two special forms of the redo request. The first special form is the null redo request. If a user receives a notification and does not wish to take any action to alter the result, then the user submits a redo request with a new_command of "none". When the system receives a null redo request, the specified failure_id is removed from the list of outstanding failure_ids. The second form is the termination redo request. Once the user has completed all compensating requests for a given failure_id, an additional redo request is submitted with a new_command of "done". When the system receives a termination redo request, the specified failure_id is removed from the list of outstanding failure_ids. All other forms of the redo command modify the costs associated with a set of indicated requests, without removing the specified failure_id from the list of outstanding failure_ids.

This design permits independent observations of the two access control mechanisms and the switching algorithm for randomizing the experiment is relatively simple to implement. There are two undesirable features of this design. First, a state of quiescence is required in order to randomly switch the behavior of the experimental system from one mechanism to the other mechanism. Quiescence may not be achieved as often as once a day. We may create a starvation state where the experimental system rarely behaves like mechanism M_k. This can result in too few sample points for mechanism M_k. Second, the users can rig the experimental system so that it cannot achieve a quiescent state. Such rigging may be an accident or an attempt to subvert the experimental system by locking it into one mode of behavior.

8.2.2 Design of an Experiment: Option 2

In our first design, the switching algorithm worked on a global basis switching the entire experimental system from using one mechanism to using another mechanism. In our second design, the switching algorithm works on mutually exclusive sets of user data objects. Each data object stored by the system has a mechanism status associated with it. The mechanism status has one of three values: SR/SW, MR/MW, or unassigned. When a request operates on object O and the mechanism status of O is unassigned the switching algorithm randomly selects the SR/SW mechanism or the MR/MW mechanism to execute

Algorithm 2 for Access Control Switching

```
For every object stored by the system Do
    Mark the object as unassigned
EndFor
For every directory entry Do
    Mark the directory entry as unassigned
EndFor
While the experiment is in progress Do
    Receive a request
    Case of request type
        Checkout x1, x2, ..., xn:
            If no xi is assigned to a method
            Then method = random selection of { SR/SW, MR/MW }
                Set the mechanism status of each xi to the method selected
                Initiate the processing of the request under the method selected
            Else If any xi is currently assigned to the SR/SW method
                Then Process this request under the SR/SW mechanism
                Else If all assigned xi are assigned to MR/MW
                    Then method = random selection of { SR/SW, MR/MW }
                        Process the request under the method selected
                EndIf
            EndIf
        Update x1, x2, ..., xn:
            Process this update under the same mechanism as its corresponding checkout
            Case of mechanism and failure status
                SR/SW and failure exists:
                    The mechanism status of all xi remains SR/SW
                MR/MW and alternate versions were created by the update:
                    The mechanism status of all xi remains MR/MW
                SR/SW and no failure exists:
                MR/MW and no alternate versions were created by the update:
                    Set the mechanism status for each xi to unassigned
            EndCase
        Redo request:
            If this is a terminating redo request
            Then Set all related mechanism statuses to unassigned
            Else Perform the request
        Requests for Directory Operations:
            If a failure exists
            Then If the specified directory entry has a mechanism status of SR/SW
                Then Process this request as an SR/SW request
                Else method = random selection of { SR/SW, MR/MW }
                    Process the request under the selected method
                    If the selected method was MR/MW
                    Then Set the mechanism status for this directory entry to unassigned
                EndIf
            Else method = random selection of { SR/SW, MR/MW }
                Process the request under the selected method
                Set the mechanism status for this directory entry to unassigned
            EndIf
    EndCase
EndWhile
```

Figure VIII.6 - Algorithm 2 for Random Selection of an Access Control Mechanism.

the request. Switching on mutually exclusive sets of data objects means that the system is quiescent from the perspective of that set of objects.

Figure VIII.6 summarizes the switching algorithm for our second design. The directory entries are treated in a manner analogous to the data objects. Each directory entry has an associated mechanism status that controls under which mechanism the request will be executed. During a failure, those data objects under the control of the SR/SW mechanism are locked into that mechanism because the system is not quiescent with respect to those data objects. These objects cannot achieve a mechanism status of unassigned while a failure exists. This preserves the single-reader and single-writer view for this set of objects. We can be more flexible with those object under the control of the MR/MW mechanism. If these objects achieve a mechanism status of unassigned during a failure they are randomly assigned to the SR/SW mechanism or the MR/MW mechanism. The MR/MW controlled data objects can become unassigned only if there are no outstanding compensating requests. Compensating requests are managed in the same manner as described in our first design.

This design permits independent observations of the two access control mechanisms but the experimental system is burdened with maintaining the mechanism status for each data object and each directory entry. Also, the switching algorithm itself is more complicated than that used in our first design. This design is superior to the first design in that the experimental system is able to switch from one mechanism to the other with greater frequency. This may allow us to gather sufficient sample points to perform statistical tests. This design, like our first design, is susceptible to rigging by the users. Accidental or intentional rigging by the users affects sets of data objects; therefore, the effects of rigging are not as acute as in our first design.

8.2.3 Design of an Experiment: Option 3

In our third design, each request that enters the experimental system is randomly assigned to be processed under one of the two access control mechanisms. No special conditions are required to switch from one mechanism to the other. The logic of each access control mechanism is performed independently while the results produced by the user requests are coerced to create a single view of the data stored by the system. By independent

processing of the access control mechanisms we mean that if a checkout and update request is assigned to execute under the SR/SW mechanism, that request can only be blocked by the requests being executed under the SR/SW mechanism. Checkout and update requests assigned to execute under the MR/MW mechanism will be processed without blocking. Directory modification requests are randomly assigned to execute under one of the access control mechanisms. For directory modifications, the model for assessing lost time is the same for both mechanisms; therefore, these requests will block each other regardless of which access control mechanism is being employeed. When modifying directory entries, the only distinction between the SR/SW mechanism and the MR/MW mechanism is that, during a failure, the SR/SW request may be rejected while the MR/MW request will be processed. Figure VIII.7 summaries the switching algorithm for our third design.

Algorithm 3 for Access Control Switching

While the experiment is in progress *Do*
    Receive a request
    Case of request type
        Update x_1, x_2, ..., x_n:
            Process the request under the mechanism that processed
                its corresponding checkout request
        Redo request:
            If the indicated requests are MR/MW requests
            Then Assess the lost time to the indicated requests
            Process the redo request
        Other Requests:
            method = random selection of { SR/SW, MR/MW }
            Record this request and the selected method
            Process the request under the selected mechanism
            Assess the lost time to this request according to the selected method
    EndCase
EndWhile

Figure VIII.7 - Algorithm 3 for Random Selection of an Access Control Mechanism.

The results of a request executed under either mechanism must be united to form one view of the data stored by the experimental system. The SR/SW access control mechanism serializes requests so that no conflicting results are produced among the requests that are processed under the SR/SW mechanism. The MR/MW access control mechanism may produce results that conflict with requests executed under either mechanism. If the results conflict with the results of another MR/MW request, then the MR/MW access control mechanism should be charged for the lost time that results from this conflict. If the results

of the MR/MW request conflict with results of an SR/SW request, then one request must be selected as the "wining" request. To always select the SR/SW request as the winning request would bias the experiment in favor of the SR/SW mechanism.

The problem of selecting winning requests is resolved as follows. If no failure exists there cannot be a conflict among the requests that modify directory entries. If a failure exists, the MR/MW mechanism may process requests that conflict with requests processed by the SR/SW mechanism. The winning requests are selected by the merge resolution algorithm. If an SR/SW request is found to be a losing request, the user is notified of the conflict. The occurrence of this conflict violates the single-reader and single-writer processing view. The user who requested the operations does not see this violation. The user knows that the experimental system may behave like a multi-reader/multi-writer system. Upon receiving a notification, the user assumes that the request was processed by the MR/MW mechanism. The user may wish to execute compensating requests. Losing SR/SW requests are not charged for the time lost by users performing compensating requests.

Conflicts between SR/SW update requests and MR/MW update requests are resolved as follows. If no failure exists and an SR/SW update request conflicts with an MR/MW update request, the first one to perform the corresponding checkout request is selected as the winning request. This means that a checkout and update request executed as a SR/SW request can result in an alternate version. The user who requested the update will be notified of the conflict. Again, the losing SR/SW request is not charged for the time lost by users performing compensating requests. If the conflict between an SR/SW update request and an MR/MW update request are discovered by a merge recovery then the winning requests are selected by the merge resolution algorithm.

This design has the advantage of employing a very simple switching algorithm, but assessing lost time to each request is more complicated than in our previous designs. To assign a cost to each request we must now keep track of all requests rather than just the MR/MW requests. Even though SR/SW requests are not charged directly for conflicts which occur, the users are not allowed to interact with a true single-reader/single-writer system. This may distort the user's perception of the SR/SW system and consequently affect the way in which the users interact with the system. The change in behavior could bias the experiment in favor of one of the mechanisms.

8.3 Gathering Sample Data

During the experiment, we gather sample points organized first by the user who submitted the request, second by the class of request submitted, and third by the access control mechanism under which the request was processed. This clustering yields the nested-factorial design [42] illustrated in figure VIII.8. Each block in the design contains a set of lost time values. We must attempt to run the experiment until a minimal number of sample points (10-12) have been taken for each of the blocks.

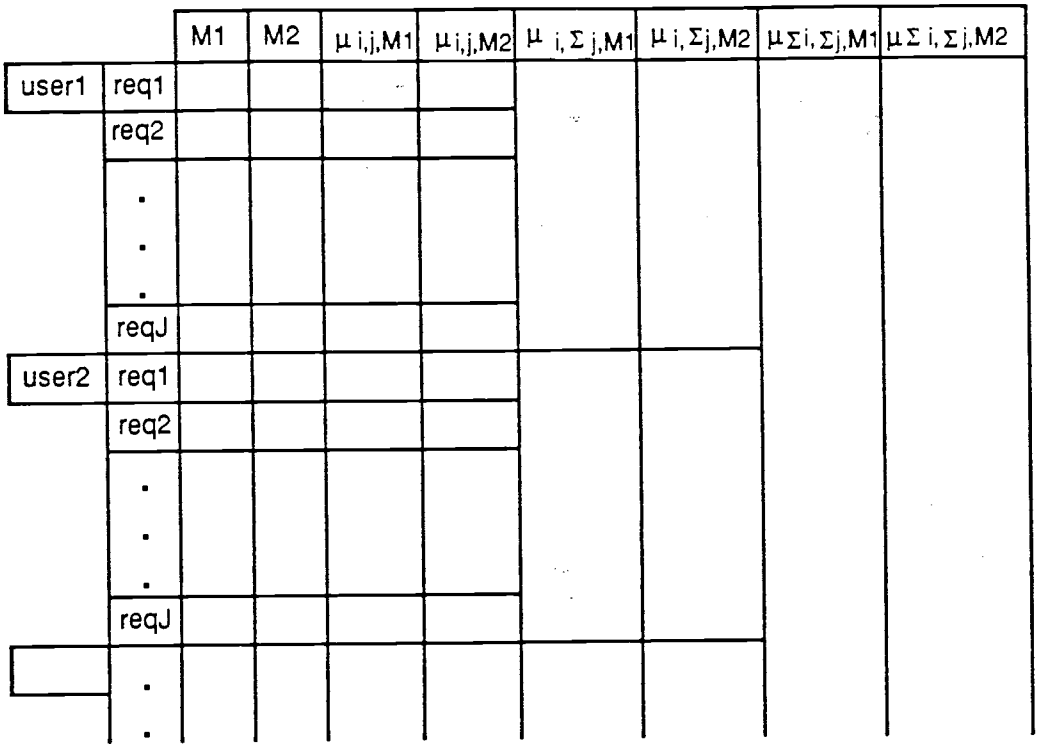


Figure VIII.8 - Design for a Nested-Factorial Experiment.

When sufficient data points have been gathered we compute the averages per user, per request class, and per mechanism. We also compute a collection of weighted averages. We can calculate a weighted average of the lost time over all users for a request class R_j executed under mechanism M_k, or a weighted average of the lost time for a user U_i over all

request classes under mechanism M_K. This design allows us to test the collection of hypotheses proposed at the beginning of this chapter.

8.4 Evaluating the Results of the Experiment

To determine whether to accept or reject each of our null hypotheses, based on the experiment, we must compare the average lost time for the SR/SW mechanism with the average lost time for the MR/MW mechanism in each category defined by our nested-factorial experiment. A one-tailed Student's t-test [3] may be used to determine whether or not the difference between the two means is statistically significant at a confidence level of up to 99%. The F-test may be used to analyze the variance among the means we have calculated for each block in the experiment. Both the F-test and the t-test assume that the sample points were drawn from normal distributions and that those distributions have equal variances. We have no a priori knowledge of the distribution to which the sample points belong. It is extremely unlikely that the lost time values will form a normal distribution. The mode of the sample points will probably be zero and the distribution will be highly right skewed. If the t-test and the F-test are to be employed, the sample points must be collected and transformed to meet the assumption of normality. Standard transformations are known for transforming common distributions to the normal distribution. Hicks [42] suggests several transformations based on relationships among the sample average, the sample variance, and the standard deviation of the original sample points. If we are unable to construct an adequate transformation to achieve an approximation of the assumptions of the t-test and the F-test, then applying these tests is not advised.

If it is determined that the t-test and the F-test should not be applied, the Wilcoxon nonparametric rank sums test [49] or the Freidman rank sums test [43] may be used to determine whether or not the variance in the two distributions is statistically significant with 96% confidence. The rank sums test assumes that the distribution from which the sample values were taken is continuous. When applying a rank test we lose the qualitative distinction between the two mechanisms we are attempting to measure. We can only conclude with 96% confidence that method A is better than method B. We cannot say how much better.

8.5 Simulation versus Experimentation

The preceding discussion of the usefulness of the multi-reader/multi-writer system versus the single-reader/single-writer system was centered on experimenting with real users in an actual work environment. Simulation represents another method for comparing and contrasting the two access control systems. Simulating the two systems allows us to calculate the response time achieved by each system, to determine where bottlenecks occur in each system, and to determine if one system is in general preferable to the other system. Each access control system should be simulated independently. The primary quantity to be measured in each simulation is the total response time required to process a sequence of requests. The primary null hypothesis investigated by such a simulation is:

H_0 For any sequence of requests R, the total response time of a single-reader/single-writer system executing R is the same as the total response time of a multi-reader/multi-writer system executing R.

The corresponding alternative hypothesis is:

H_1 For any sequence of requests R, the total response time of a single-reader/single-writer system executing R is greater than the total response time of a multi-reader/multi-writer system executing R.

The queueing model proposed for each system is inherently tied to the implementation details of each access control mechanism. The model must capture the processing structure and the processing delays unique to each of the mechanisms.

8.5.1 Simulating a Single-Reader/Single-Writer System

Figure VIII.9 presents a closed world simulation model for a lock based single-reader/single-writer access control system. Circles and ovals represent server processes, rectangles represent queues for those processes, and the arrows show the possible flow of a request being processed by the system. The queueing system illustrated represents a single site in the distributed system. The model is repeated once for each simulated site. A single site runs on a single central processing unit which is shared by all server processes on that

site; therefore, the server processes on a single site are always run in series. The running processes, one per simulated site, are run in parallel.

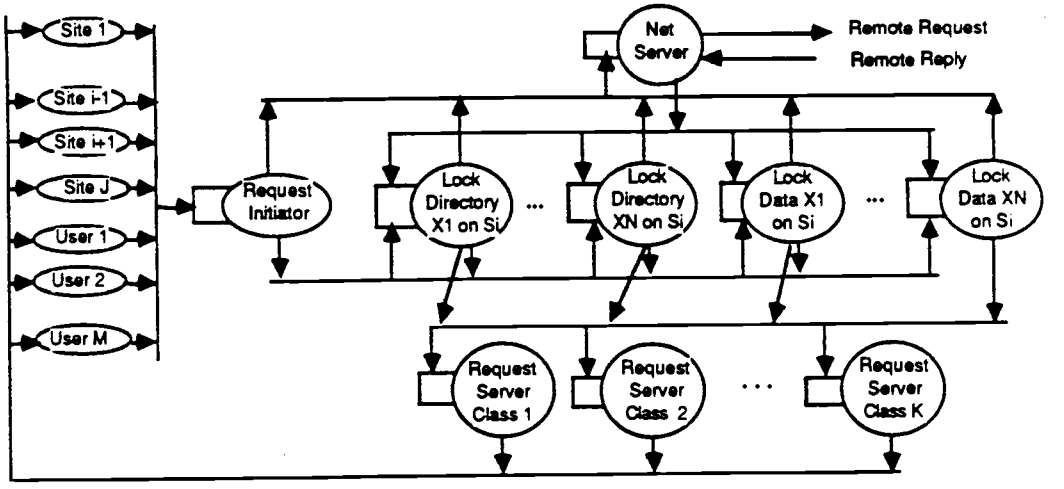


Figure VIII.9 - Queueing Model for the Single-Reader/Single-Writer System.

Each of the model processes has an associated service time. Service times for the lock processes, the request servers, and the net server can be estimated by making preliminary time measurements on an existing lock based SR/SW. These are deterministic processes. User processes are more complicated general processes. Estimating service times for a general process is difficult. The three experiments described above could be used as preliminary studies of a user process. From these preliminary studies, we can construct one or more models of a user process. In particular, the experiments would allow us to determine a range of service times for a user process.

Each connection in the queueing model has an associated probability. This is the probability that that path will be traversed when processing a request. These probabilities are used to calculate the rate at which requests arrive at each queue. The arrival rate for a given queue is a function of the service time of the set of processes which add entries to that queue. The service times of the user processes govern the arrival rate for the initial request queue. All other arrival rates are dependent on this initial arrival rate.

The simulated system models the following behavior. Simulated users submit requests to the simulated system. These requests are queued awaiting processing by the request

```
Algorithm for SR/SW Lock Process

If the queue entry is a network reply
Then If the corresponding request requires more locks
        Then Forward the request to the queue of the next directory
                                or data entry to be locked
        Else Forward the request to the appropriate request server
Else If the directory entry is not currently locked
        Then Lock this directory entry
                If the request requires more locks
                Then If the local site is the lock server for the next item to be locked
                        Then Forward the request to the queue of the next directory
                                                or data entry to be locked
                        Else Forward the request to the net server
                Else Forward the request to the appropriate request server
                EndIf
        EndIf
EndIf
```

Figure VIII.10 - Algorithm for Lock Processes in a SR/SW Model.

initiator. This initial request queue is serviced on a first-come-first-serve basis. The request initiator translates each request into an ordered list of locks that must be obtained before the request can be processed. The request initiator forwards the request to the net server queue or a local lock server queue so that the request may obtain its first lock. The lock servers of the SR/SW model form an ordered series of processes. Each simulated site S_i has a lock server for each storage system directory entry and each data object controlled by S_i. Locks that cannot be obtained from the locally simulated site are acquired by the net server from simulated remote sites. Each lock process queue may contain lock requests and net server replies. The lock requests are processed on a first-come-first-serve basis and the net server replies pre-empt all other queued requests by moving immediately to the head of the queue. Figure VIII.10 shows the processing algorithm of the lock processes in the SR/SW model. When a lock process has serviced a request, that request is forwarded to the next process whose service is required. If the request needs another lock, the request is forwarded to a local lock process or the net server. If all required locks have been obtained, the request is forwarded to the appropriate request server. The request queue for the request server is serviced on a first-come-first-serve basis. The request server carries out the requested action and releases all of the locks held by the request. The model does not simulate broadcasting the results of the request to all simulated sites. This part of the processing would be carried out in identical fashion by both of the access control mechanisms; therefore, we have

removed this from the SR/SW queueing model and the MR/MW queueing model. Once the request has been processed, the request and its reply are returned to the requesting user. The user is then free to initiate another request.

8.5.2 Simulating a Multi-Reader/Multi-Writer System

Figure VIII.11 presents a closed world simulation model for a lock based multi-reader/multi-writer access control system. Circles and ovals represent server processes, rectangles represent queues for those processes, and the arrows show the possible flow of a request being processed by the system. This model represents the series of processes running on a single site. The model is replicated for each simulated site.

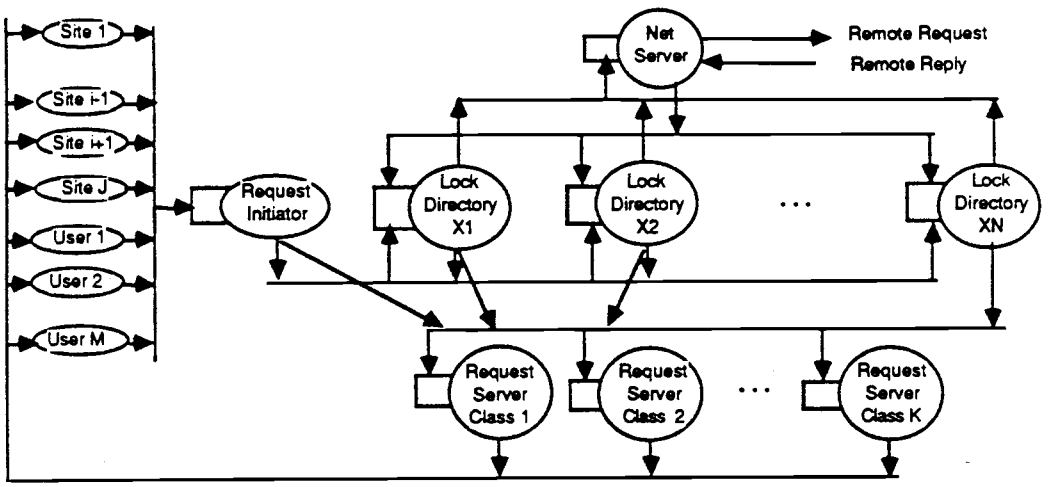


Figure VIII.11 - Queueing Model for the Multi-Reader/Multi-Writer System.

Each of the processes has an associated service time and each connection has an associated probability which provides estimated arrival rates for each queue. The service times and arrival rates for the MR/MW system differ from the service times and arrival rates in the SR/SW system. Service times can be estimated by performing preliminary measurements of the service time of the processes in our prototype MR/MW system. The service time ascribed to a user process must be estimated by other means. The DHSS

prototype was used by a small group of engineers for a three week period to store documents which were written as a team effort. The number of users and the duration of usage were insufficient to develop a model of a user process in the MR/MW environment. One or more of the larger experiments proposed in this chapter should yield sufficient information to construct a model of a user process in the MR/MW environment and allow us to estimate the range of service times ascribed to such a user process.

Algorithm for MR/MW Locker Process:

```
If the queue entry is a network reply
Then If the corresponding request requires more locks
        Then Forward the request to the queue of the next directory entry to be locked
        Else Forward the request to the appropriate request server
Else If the local site is not currently the lock server for this directory entry
        Then Forward the lock request to the net server
        Else If this directory entry is not currently locked
                Then Lock this directory entry
                        If the request requires more locks
                        Then Forward the request to the queue of the next directory
                                        entry to be locked
                        Else Forward the request to the appropriate request server
                        Endlf
                Endlf
        Endlf
Endlf
```

Figure VIII.12 - Algorithm for Lock Processes in a MR/MW Model.

The simulated MR/MW system models the following behavior. Simulated users submit requests to the system. These requests are queued awaiting processing by the request initiator. This initial request queue is serviced on a first-come-first-serve basis. The request initiator translates each request into an ordered list of locks that must be obtained before the request can be processed. If no directory locks are required, the request is forwarded to the appropriate request server queue. The MR/MW model contains a lock process for each storage system directory entry even though the local site may not be the lock site for every directory entry. Figure VIII.12 shows the processing algorithm of the lock processes in the MR/MW model. If the local site is the locking site for the specified directory entry then the lock process will grant the lock locally. If the local site is not the locking site for the specified directory entry then the lock process forwards the request to the net server so that the lock may be obtained from a remote lock process. The queue for each lock server may contain lock requests and network replies. The network replies pre-empt all other queued

request by moving immediately to the head of the queue and the lock requests are added at the end of the queue. Once all required locks have been obtained, the request is forwarded to the appropriate request server. The request server performs the requested action, releases any locks held by the request, and returns the request and the reply to the user.

8.5.3 Measuring the System

To accept or reject our primary null hypothesis we must measure the total response time required by each system to process identical sequences of requests. The difference in total response time should not be considered to be significant unless the difference is at least five to ten percent of the total response time of the slower system. If the difference is less than five percent of the total response time, it is unlikely that the users will be able to detect any difference in the overall performance of the two systems. If one mechanism is significantly superior to the other for almost every sequence of requests, we may safely reject our null hypothesis.

Simulation as a tool allows us to observe many properties of the system being simulated. Factors such as the average queueing delay experienced by a request, the average queue size, the maximum busy time and the maximum idle time for a server, and the average number of requests in the system at any one time may be measured during a simulation. For systems with a large initial arrival rate, these observations aid in locating bottlenecks in the current system design. System designers may use this information in designing a better system. As we endeavor to improve the processing structure of the MR/MW system this information would be useful.

CHAPTER NINE
CONCLUSIONS AND FUTURE WORK

The goal of this research has been to design a robust distributed storage system for use in an engineering design database system. In chapter one, we presented the many distinctions between a design environment and a traditional data processing environment. The key distinctions include the need for storing alternate versions of a design, the large amount of time between reading and updating, and the large number of modifications installed by a single update. These distinctions motivated us to investigate optimistic access control protocols (multi-reader/multi-writer) for use in a distributed engineering design database system. Given the specification for a versioned storage model, it became clear that optimistic access control could be employeed even during failures, but such usage would require a special recovery mechanism to compliment the optimistic access control. Combining optimistic access control and its complimentary recovery protocols we designed and prototyped a robustness mechanism for the Distributed Hypothetical Storage System.

9.1 Summary of Results

In chapter 4 we defined the optimistic access rule which allows almost all permitted requests to be executed at any time. To make this an effective rule we designed algorithms for the creation and destruction of pseudo name server sites and pseudo primary sites. In lemma 5 we demonstrated that pseudo primary sites combined with our locking protocol ensure that no client requests can be hung indefinitely; thus these mechanisms effectively implement the optimistic access rule.

We used the concept of a virtual partition to represent a set of communicating sites working together. We presented a divergence recovery mechanism for managing site crash and communication failure, and a merge recovery mechanism for managing media failure, the recovery of a crashed site, the correction of a communication failure, and the introduction of new sites into the system. By the results of chapters 3, 4, 5, and 6 we demonstrated that these recovery mechanisms maintain mutual consistency among the sites

in a virtual partition. The maintenance of mutual consistency is a fundamental requirement of any distributed system.

9.2 Observations and Insights

While designing the robustness protocols we made four interesting observations or insights. In DHSS, we designed a protocol for divergence recovery and a protocol for merge recovery. Most systems make no distinction between divergence and merge. Making no distinction between divergence and merge would simplify the recovery mechanisms required, but in DHSS this simplification was not possible. In DHSS we must distinguish between divergence and merge because we must provide for the detection and resolution of conflicts when a merge occur. When a divergence occurs, there are no conflicts to be detected. This implies that managing a failure is different from managing the correction of a failure.

All of the failure detection protocols we designed for DHSS are based on the use of timeouts. When implementing a robustness mechanism, selecting timeout values is very difficult. Other researchers involved in prototyping distributed systems claim that timeouts are unreliable and should be avoided whenever possible [74]. Avoiding the use of timeouts is not feasible. Without timeouts a hung site that has not crashed will hang all sites attempting to communicate with it. Therefore, timeout based protocols are essential. Achieving a reasonable implementation of timeout based protocols may require defining a set of timeouts, so that appropriate timeout factors may be used in different situations.

The most elegant feature of recovery in DHSS is that merge recovery provides a unilateral resolution of conflicts. The ability of each site to compute the same resolution without negotiating reduces the cost of performing resolution. If the resolution of each detected conflict required a k-phase negotiation, such a k-phase protocol would be inherently complex. Also the k phases would be carried out over the network, making merge recovery even more time consuming. Such k-phase negotiations are probably not suitable in a workstation environment.

After two years of designing and refining the recovery protocols, we have concluded that the essence of the problem of recovery lies in managing those requests that are caught in the gap between stable processing states. These requests are the ones that discover remote failures and the ones whose processing is interrupted by a local failure. If processing is momentarily suspended when a failure is detected, the number of requests caught in the failure gap is very small. Maintaining an internally consistent and mutually consistent state with respect to the outcome of these few requests and minimizing the number of requests that fall into the failure gap requires an enormous and complex effort in terms of recovery protocols. The complexity is a result of the asynchrony of the distributed environment.

9.3 Future Work

Our future work with DHSS is divided into two categories. First, we would like to carry out one of the experiments defined in chapter eight of this thesis. Under the assumption that failures do occur and that users attempt to use the system while failures exist, it is clear that the multi-reader/multi-writer system provides better throughput than the single-reader/single-writer system. We believe that a multi-reader/multi-writer system increases productivity in general. The main results of this thesis are that multi-reader/multi-writer systems can be designed and that they will function properly. The goal of performing this experiment is to substantiate that it is worthwhile to build such a system.

Second, we would like to make enhancements to the system. Many interesting DHSS enhancements have been proposed. Most of these enhancements have little or no effect on the robustness protocols for DHSS. There is one major enhancement to DHSS that would affect the robustness mechanism. We would like to augment the storage system so that it would perform automatic merging of conflicting updates whenever possible.

DHSS has no understanding of the data it stores for its clients. In order to perform automatic data merges, DHSS must be aided by a set of tool programs that would be used to test whether or not the updates could be merged and if so, to perform the actual merge on the client's data. The tool programs would encapsulate the semantics of the client's data. DHSS would possess the very limited semantic knowledge of what tools should be applied to what objects.

It is not always possible to merge the modifications performed by two or more conflicting updates. From an abstract view, each object version consists of a collection of basic units of information. If two modifications set one basic unit to two different values, those modifications cannot be merged. When two modifications alter the value of distinct units within a version it may or may not be possible to merge the modifications. Two such modifications could be logically inconsistent. Suppose the data being stored is program source code and the basic unit is a program statement. Consider the sequence of statements "x = 0", "y = 1", "z = 0", "assert $x + y + z \leq 2$". One client could modify the assignment of the variable x to be "x = 1". A second client could independently modify the assignment of the variable z to be "z = 1". Merging these two modifications results in the logically inconsistent sequence of statements "x = 1", "y = 1", "z = 1", "assert $x + y + z \leq 2$". The test tool processing these conflicting versions would declare that the two versions cannot be merged.

To make use of the test and merge tools, DHSS would associate a type with each DHSS object. This type would allow the storage system to conclude which tool should be executed to test whether or not a true merge of the set of conflicting updates is possible. If a true merge is possible, the object type would specify which merge program should be executed to carry out the data merge. DHSS would add the merged data as the real update. The original conflicting updates would be stored as alternate versions of the object. Retaining the original conflicting updates as alternate versions is desirable for two reasons: first, the users may have intended to create one or more alternate versions; second, after making modifications to the merged version, the users may decide that they wish to pursue a new version based on only one of the conflicting versions. If the conflicting versions were discarded, the users would be unable to use those versions for any future purpose.

Figure IX.1(A) shows conflicting updates to an object Obj1 performed in separate virtual partitions. Figure IX.1(B) shows the result of merge recovery if the conflicting updates could be merged by an auxiliary tool process. The clients would be notified of the conflict and the merged resolution.

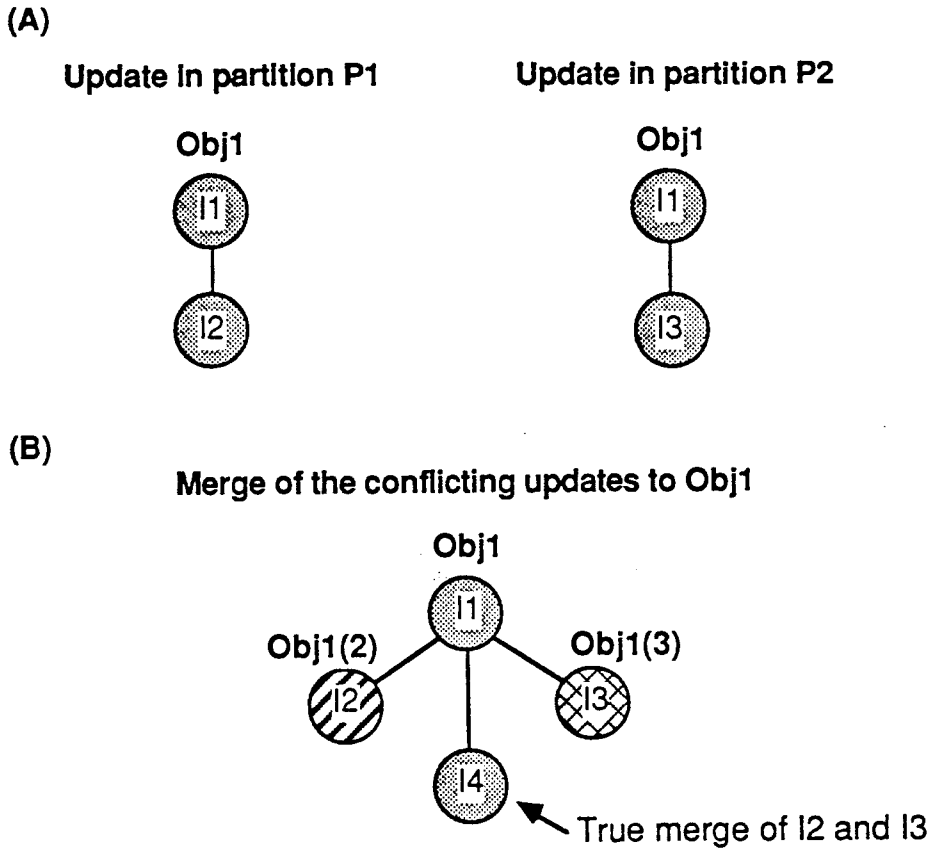


Figure IX.1 - Merging Conflicting Data using Auxiliary Tools.

BIBLIOGRAPHY

In the following references, ACM stands for the Association for Computing Machinery, IEEE stands for the Institute of Electrical and Electronics Engineers, SIGACT stands for the Special Interest Group for Automata and Computability Theory, SIGMOD stands for the Special Interest Group for the Management of Data, COMPCON stands for the Computer Society International Conference, LBL stands for Lawrence Berkeley Laboratory, and IBM stands for International Business Machines.

References

1. P. Alsberg and J. Day, "A Principle for Resilient Sharing of Distributed Resources," in *Proceedings of the Second International Conference on Software Engineering*, 1976.

2. T. Anderson, E. Ecklund, and D. Maier, "PROTEUS: Objectifying the DBMS User Interface," in *1986 International Workshop on Object-Oriented Database Systems*, pp. 133-145, September 1986.

3. H. Arkin and R. Colton, *Statistical Methods, Fifth Edition*, Barnes & Noble Books, 1970.

4. R. Attar, P. Bernstein, and N. Goodman, "Site Initialization, Recovery, and Backup in a Distributed Database System," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 645-649, November 1984.

5. C. Attiya, D. Dolev, and J. Gil, "Asynchronous Byzantine Consensus," in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 119-133, August 1984.

6. T. Atwood, "An Object-Oriented DBMS for Design Support Applications," in *Proceedings of Computer Aided Technologies COMPINT 85*, IEEE, Montreal, Quebec, Canada, September 1985.

7. B. Awerbuch and S. Even, "Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network," in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 278-281, August 1984.

8. C. Bachman, "Integrated Data Store," *Data Processing Management Association Quarterly*, pp. 61-80, January 1965.

9. D. Badal, "Concurrency Control Overhead or Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms," in *Proceedings of the Fifth Berkley Conference on Distributed Data Management and Computer Networks*, pp. 85-104, 1981.

10. P. Bernstein, N. Goodman, J. Rothnie, and C. Papadimitriou, "Analysis of serializability of SDD-1: a system for distributed databases (the fully redundant case)," *IEEE Transaction on Software Engineering*, vol. 4, no. 3, pp. 154-168, 1978.

11. P. Bernstein and N. Goodman, "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," in *Proceedings of the Sixth International Conference on Very Large Data Bases*, pp. 285-300, Montreal, Canada, October 1980.

12. P. Bernstein, D. Shipman, and J. Rothnie, "Concurrency control in a system for distributed databases (SDD-1)," *ACM Transactions on Database Systems*, vol. 5, no. 1, pp. 18-51, 1980.

13. L. A. Bjork, Jr., "Generalized audit trail requirements and concepts for data base applications," *IBM Systems Journal*, vol. 14, no. 3, pp. 229-235, 1975.

14. CADTEC, "CORD™ Data System Overview," in *CORD System Overview Manual*, January 30, 1985.

15. R. Casey, "Allocation of copies of a file in an information network," in *Proceedings of the American Federation of Information Processing Spring Joint Computer Conference*, pp. 617-625, 1972.

16. S. Ceri, G. Martella, and G. Pelagatti, "Optimal File Allocation for a Distributed Data Base on a Network of Minicomputers," in *International Conference on Data Bases*, pp. 216-237, Heyden, 1980.

17. S. Ceri and S. Owicki, "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," in *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 117-129, February 1982.

18. D. Chamberlin, "Relational Data-Base Management Systems," *Computing Surveys*, vol. 8, no. 1, pp. 43-66, March 1976.

19. A. Chan, S. Fox, W. Lin, A. Nori, and D. Ries, "The Implementation of an Integrated Concurrency Control and Recovery Scheme," in *Proceedings of the ACM SIGMOD 1982 International Conference on Management of Data*, pp. 184-191.

20. W. Chu, "Optimal file allocation in a multiple computer system," *IEEE Transactions on Computers*, vol. C-18, no. 10, pp. 885-889, 1969.

21. J. Clifford and A. Tansel, "On An Algebra For Historical Relational Databases: Two Views," in *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data*, pp. 247-265, May 1985.

22. E. F. Codd, "A relational model for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, 1970.

23. E. F. Codd, "Extending the Database Relational Model to Capture More Meaning," *ACM Transactions on Database Systems*, vol. 4, no. 4, pp. 397-434, December 1979.

24. R. A. Crus, "Data Recovery in IBM Database 2," *IBM Systems Journal*, vol. 23, no. 2, pp. 178-188, 1984.

25. S. Davidson, "Optimism and Consistency in Partitioned Distributed Database Systems," *ACM Transactions on Database Systems*, vol. 9, no. 3, pp. 456-482, September 1984.

26. D. Dolev, S. Dwork, and L Stockmeyer, "On the Minimal Synchronization Needed for Distributed Consensus," in *Proceedings of the Twenty-fourth Annual Symposium on Foundations of Computer Science*, pp. 393-402, 1983.

27. C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 103-118, August 1984.

28. D. L. Eager and K. C. Sevcik, "Achieving Robustness in Distributed Database Systems," *ACM Transactions on Database Systems*, vol. 8, no. 3, pp. 354-381.

29. D. Ecklund, "Object Management System User's Manual," Tektronix Laboratories Internal Technical Report CR-84-33, December 21, 1984.

30. D. Ecklund, "Robustness in Distributed Hypothetical Databases," in *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*, pp. 643-656, 1986.

31. E. Ecklund, D. Price, R. Krull, and D. Ecklund, "Federations: Scheme Management in Locally Distributed Databases," in *Proceedings of the Nineteenth Hawaii International Conference on System Science*, pp. 395-407, 1986.

32. E. F. Ecklund, Jr. and D. M. Price, "Multiple Version Management of Hypothetical Databases," in *Proceedings of the Eighteenth Hawaii International Conference on System Sciences*, pp. 163-173, 1985.

33. A. ElAbbadi, D. Skeen, and F. Cristian, "An Efficient, Fault-Tolerant Protocol for Replicated Data Management," in *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 215-229, Portland, Oregon, March 25-27, 1985.

34. A. ElAbbadi and S. Toueg, "Availability in Partitioned Replicated Databases," in *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 240-251, March 1986.

35. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624-633, November 1976.

36. D. Gifford, "Weighted Voting for Replicated Data," in *The Seventh Symposium on Operating Systems Principles*, pp. 150-162, ACM, 1979.

37. V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 435-439, September 1980.

38. A. Goldberg and D. Robson, *Smalltalk-80 The Language and Its Implementation*, Addison Wesley, 1983.

39. J. Gray, "Notes on Data Base Operating Systems," in *Operating Systems An Advanced Course*, ed. R. Bayer, R. Graham, and G. Seegmüller, pp. 393-481, Springer-Verlag, 1978.

40. J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The Recovery Manager of the System R Database Manager," *Computing Surveys*, vol. 13, no. 2, pp. 223-242, June 1981.

41. Data Base Task Group, *CODASYL Data Base Task Group April 71 Report*, Association for Computing Machinery, New York, 1971.

42. C. Hicks, *Fundamental Concepts in the Design of Experiments, Third Edition*, Holt, Rinehart and Winston, 1982.

43. M. Hollander and D. Wolfe, *Nonparametric Statistical Methods*, John Wiley & Sons, 1973.

44. J. Hopcroft and J. Ullman, *Formal Languages and Their Relation to Automata*, Addison-Wesley Publishing Company, 1969.

45. C. Jullien, A. Leblond, and J. Lecourvoisier, "A Database Interface for an Integrated CAD System," in *Proceedings of the Twenty-third ACM/IEEE Design Automation Conference*, pp. 760-767, June 1986.

46. R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, ed. R. E. Miller and J. W. Thatcher, pp. 85-103, Plenum Press, New York, 1972.

47. R. Katz, "Transaction Management in the Design Environment," in *Proceedings of the Second International Conference on Databases*, pp. 259-273, Heyden, September 1983.

48. R. Katz, M. Anwarrudin, and E. Chang, "A Version Server for Computer-Aided Design Data," in *Proceedings of the Twenty-third ACM/IEEE Design Automation Conference*, pp. 27-33, June 1986.

49. R. Kirk, *Experimental Design: Procedures for the Behavioral Sciences*, Brooks/Cole Publishing Company, Wadsworth Publishing, 1968.

50. H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213-226, 1981.

51. H. Kuss, "On Totally Ordering Checkpoints in Distributed Data Bases," in *Proceedings of the ACM SIGMOD 1982 International Conference on Management of Data*, pp. 293-302.

52. G. Le Lann, "Algorithms for distributed data-sharing systems which use tickets," in *Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Network*, pp. 259-272, August 1978.

53. D. Leblang and G. McLean, Jr., "Configruation Management for Large-Scale Software Development Efforts," in *Workshop of Software Engineering Environments for Programming-in-the-Large*, pp. 122-127, Harwichport, Massachusetts, June, 1985.

54. R. Lorie, "Physical Integrity in a Large Segmented Database," *ACM Transactions on Database Systems*, vol. 2, no. 1, pp. 91-104, March 1977.

55. S. Mahmoud and J. Riordon, "Optimal Allocation of Resources in Distributed Information Networks," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 66-78, March 1976.

56. W. McGee, "The Information Management System IMS/VS Part I: General structure and operation," *IBM Systems Journal*, vol. 16, no. 2, pp. 84-95, 1977.

57. W. McGee, "The Information Management System IMS/VS Part II: Data base facilities," *IBM Systems Journal*, vol. 16, no. 2, pp. 96-122, 1977.

58. D. McLeod, "An Object-Oriented Approach to Databases for VLSI-CAD," in *Workshop on Information System Support for Integrated Design and Manufacturing Processes*, 1986.

59. D. Menasce and R. Muntz, "Locking and Deadlock Detection in Distributed DataBases," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 195-202, May 1979.

60. R. Metcalfe and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, vol. 19, no. 7, pp. 395-403, July 1976.

61. H. Morgan and J. Levin, "Optimal Program and Data Locations in Computer Networks," *Communications of the ACM*, vol. 20, no. 5, pp. 315-322, 1977.

62. Z. M. Ozsoyoglu and G. Ozsoyoglu, "An Extension of Relational Algebra for Summary Tables," in *Proceedings of the Second LBL Workshop on Statistical Database Management*, 1983.

63. G. Popek, G. Thiel, and C. Kline, "Recovery of Replicated Storage in Distributed Systems," UCLA Computer Science Technical Report, 1983.

64. D. Price and D. Maier, "Data Model Requirements for Engineering Applications," in *Proceedings of the International Workshop on Expert Database Systems*, ed. L. Kerschberg, 1984.

65. D. Reed, "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems*, vol. 1, no. 1, pp. 3-23, 1983.

66. A. Reuter, "A Fast Transaction-Oriented Logging Scheme for UNDO-Recovery," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 4, pp. 348-356.

67. M. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364-369, December 1975.

68. D. Rosenkrantz, R. Stearns, and P. Lewis II, "System level concurrency control for distributed data base systems," *ACM Transactions on Database Systems*, vol. 3, no. 2, pp. 178-198, 1978.

69. S. Sarin, B. Blaustein, and C. Kaufman, "System Architecture for Partition-Tolerant Distributed Databases," *IEEE Transaction on Computers*, vol. C-34, no. 12, pp. 1158-1163, December 1985.

70. D. Severance and G. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Transactions on Database Systems*, vol. 1, no. 3, pp. 256-267, September 1976.

71. R. Sprague and H Watson, "MIS Concepts - Parts I and II," in *Interactive Decision Oriented Data Base Systems*, ed. W. House, Petrocelli/Charter, 1977.

72. M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 188-194, May 1979.

73. R. Taylor and R. Frank, "CODASYL Data-Base Management Systems," *Computing Surveys*, vol. 8, no. 1, pp. 67-103, March 1976.

74. M. Templeton, *Personal Communication*, January 23, 1986.

75. R. Thomas, "A solution to the concurrency control problem for multiple copy data bases," in *IEEE COMPCON*, pp. 56-62, February 1978.

76. W. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the Sixth International Conference on Software Engineering, IEEE*, Tokyo, Japan, September 1982.

77. D. Tsichritzis and F. Lochovsky, "Hierachical Data-Base Management: A Survey," *Computing Surveys*, vol. 8, no. 1, pp. 105-123, March 1976.

78. J. Ullman, *Principles of Database Systems, Second Edition*, Computer Science Press, 1982.

79. B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," in *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pp. 49-70, October 1983.

80. B. J. Walker, "Issues of Network Transparency and File Replication in the Distributed Filesystem Component of LOCUS," Ph.D. Dissertation University of California at Los Angeles, 1983.

81. S. Weiss, K. Rotzell, T. Rhyne, and A. Goldfein, "DOSS: A Storage System for Design Data," in *Proceedings of the Twenty-third ACM/IEEE Design Automation Conference*, pp. 41-47, June 1986.

APPENDIX

APPENDIX I - Operations Supported by DHSS

This appendix contains a discussion of the operations supported by the DHSS from a DHSS client's point of view. The message format for each request type is given using the following notation: * means zero or more repetitions; + means one or more repetitions; groupings are enclosed in parentheses ( ); and optional words are enclosed in square brackets [ ].

Many of the commands refer to names of objects stored by the storage system. These names must be in the form of an object base identifier. An object base identifier must be of one of the following forms: entity_name or entity_name[time]. A valid entity_name must be of one of the following forms: object_name, alternate_path_name, or object_name(alias#). An object_name and and alternate_path_name are decomposable into three parts: user-defined name, name of the user that created the user-defined name, and the name of the site on which the user-defined name was created.

Open_Object_Base

open_object_base [federation_name]

The open_object_base command is used to initiate a session with the DHSS. This step must be performed before any other DHSS requests are made. The federation name is optional. If one is supplied then an open_federation command is also executed. This allows session initiation and opening of a federation in one request. The federation name specified must be the name of an existing federation.

Close_Object_Base

close_object_base

The close_object_base command is used to terminate a session with the DHSS. If the user has failed to close the most recently opened federation, it is closed automatically.

Define

define federation_name [replication_factor]

The define command is used to create a new federation on the local site. The specified federation name must be unique over all federation names known at the local site. When a user defines a new federation, the local site must supply sufficient storage to maintain a copy of the system directory and storage for copies of data instances created by the federation members on the local site. Initially the federation contains zero objects. The defining user is by default a participating member in the federation, that is, the user can create and own data objects in the new federation.

Enroll

enroll federation_name site_name participation_mode

The enroll command is used to become a member of an existing federation. There may be several federations having the same name, thus the desired federation must be specified by its name and a site that currently belongs to that federation.

Valid participation modes are participant and associate. A participating member may create and own any number of data instances in the federation. Actions restricted to owners may be performed by the owning participating member only. Also permission may be granted to a participating member for performing actions on data instances owned by other participating members. An associate member may not create or own any data instances in the federation. Associate members may read any data instances that participating members have permitted them to read.

All sites must provide sufficient secondary storage for maintaining the storage system directory. Sites with local participating users must also provide storage for each data instance that is owned by a local user.

Open_Federation

open_federation federation_name

The open_federation command is used to designate a context for mapping entity names. All subsequent commands will be interpreted as referring to objects in the specified federation. The user issuing the command must be a member of the specified federation and the federation must exist.

Close_Federation

close_federation

The close_federation command is used to terminate the current context for mapping entity names in subsequent DHSS requests.

Checkout

checkout (object_base_name filename)+

The checkout command is used to provide the client with a readable copy of the requested data. The checkout request must specify a set of valid object base names in the name space of the open federation. DHSS maps the set of names to a set of data instances and retrieves a readable copy of each instance to the filename specified. When data is acquired by a checkout, DHSS assumes that the data will be copied and modified by the client, and eventually stored in the database via an update request. Information on who requested the checkout, what data was involved, and the status of each data instance is maintained by DHSS so that configuration consistency can be maintained when the associated update request is performed. The possible status values of an instance at checkout time are current of a principal path, current of an alternate path, and non-current.

Return_Data

return_data (object_base_name filename)+

The return_data command is used to release data that has been obtained by a checkout request but will not be updated. This causes DHSS to discard all information about the associated checkout. The filenames specified should be those names specified in the associated checkout request.

Read_data

read_data (object_base_name filename)+

The read_data command is used to obtain access to data instances for the purpose of browsing. Data obtained by the read request is not updatable. DHSS does not maintain any information on who or what data was acquired by a read_data request.

Create

create object_name source_filename [replication_factor]

The create command is used to create a new DHSS object in the open federation. The proposed object name must be unique within the open federation and the source file must exist. The source file contains the data to be used as the initial instance for the new object. The replication factor specifies the number of copies to store of each instance in this DHSS object. If no replication factor is specified the replication factor for the open federation is used.

If the object name is acceptable, the storage server will copy the initial data value to stable storage on the local site, add new entries for this object to the system directory, and make the required number of copies of the data. The newly created object logically consists

of a single instance that belongs to two paths, the principal path and the path having implicit alias one. The requesting user will be the owner of the object. All access permission for the new object is access by owner only.

Derive

derive object_base_name [alternate_path_name]

The derive command creates an alternate version in an existing object. The instance corresponding to the object_base_name specified becomes the current instance of the new alternate path. The new path will be assigned an implicit alias number unique within the effected DHSS object. Optionally a name may be proposed for the new path. If a path name is proposed it must be unique within the open federation. The requesting client becomes the owner of the new path and is notified of its implicit alias number.

Update

update (object_base_name filename [replication_factor])+

The update command causes a set of new temporal versions to be stored in the database. Each object_base_name specifies a path to be updated and the associated filename is the name of the file containing the new data instance to be stored as the current version of the specified path. The optional replication factor specifies the number of copies to make of the new temporal version.

Each update request must have been preceded by a checkout request for the exact set of paths to be updated. The DHSS will compare the current status of each instance with its status at the time of the checkout. If all statuses are the same then all updates are applied creating new temporal versions in the paths specified by the update. If any of the statuses have changed since the time of the checkout, some or all of the updates will result in implicit derives followed by an update to the newly derived path. All implicitly derived paths are

owned by the requesting client. The client is notified of all late updates and the implicit alias numbers generated for the new paths.

Late updates cause implicit derives according to the following rules.

1. If all instances are the current version of a principal path then apply the updates to the principal paths producing new temporal versions of each object in the group.

2. If all instance are the current version of an alternate path then apply the updates to the alternate paths producing new temporal version of each alternate path in the group.

3. If some subgroup of the instances are the current version of an alternate path and the other instances are non-current versions of their respective paths, then for each of the non-current instances create and substitute a new alternate path rooted at that instance and carry out the updates according to rule 2.

4. If some, but not all, of the instances are the current versions of a principal path, then for each instance that is the current of a principal path create and substitute a new alternate path rooted at that instance and carry out the updates according to rule 3.

Delete

delete object_name

The delete command is used to remove all instances of a DHSS object. Only the object owner may delete the object. All paths within the object are deleted regardless of the ownership of the individual path. The name of a deleted object may not be immediately available for re-use as a unique name because physical deletion may be slightly delayed. The object is logically deleted at the completion of the delete request.

Erase

erase entity_name option

The erase request removes instances from paths. The two options for erasing are erase one and erase all. The erase one request deletes the current instance in a path reverting to the previous instance as the current. The path specified by the entity_name must contain more than one temporal version.

The erase all request will remove all instances in a single alternate path. The principal path cannot be erased. The name of the entity specified in the erase all request must be the name of a path. The name of the erased path may not be immediately available for reuse as a unique name because physical deletion may be slightly delayed. The implicit alias number corresponding to the erased path will never be reused.

Execution of an erase one or an erase all does not result in the loss of instances that were shared by non-erased paths.

Name

name entity_name new_name

The name command is used to replace the current name of a DHSS object or the current name of an alternate path within an object. The name command may also be used to give a name to a path that is currently named only by the object to which it belongs and the path's implicit alias. (An unnamed path is created by the execution of a derive command with no specified alternate path name or by the execution of a late update.) The proposed new name must be unique within the federation thus maintaining a flat name space. At any given time, an object or a path may have only one name from the federation's name space.

Name_Match

name_match user-defined_name

The name_match command queries the system directory and retrieves a list of names matching the specified user-defined name. The returned list will contain fully modified names consisting of user-defined name, user name, and site name, separated by the symbol "^". a list of full names where the user-defined

Assign

assign object_name alternate_path_name

The assign command is used to alter the mapping of a DHSS object name in a manner contrary to the semantics of update. The assign request will force an object name to be mapped to the current instance of a specified path in the tree of instances. Semantically the assign allows the client to select a related alternate version of an object as the principal version of that object.

New_Owner

new_owner entity_name user_name site_name

The new_owner request transfers ownership of a DHSS object or a path within an object to a specific user at a specific site. The proposed new owner must be a participating member of the federation. All privileges and responsibilities of ownership are transferred to the new owner. Only the current owner can execute a new_owner request.

Usurp

usurp alternate_path_name

The usurp command is used by the owner of a DHSS object to usurp ownership of a path that was derived from an instance of the object. The path owner is powerless against dictatorial object owners. The new owner possesses all privileges and responsibilities of ownership.

Set_Permission

set_permission entity_name operation permission_setting (user_name site_name)*

DHSS maintains access permissions for each DHSS object and each path within an object on a per operation basis. The set_permission command is used to alter a general permission setting for a specific operation on an object or an alternate version of an object. The operations for which permissions are maintained are assign, checkout, update, derive, name, read_data, set_permission, grant, deny, and global. Global means all of the permissible operations. The general permission settings are private, public, private-include, or public-exclude. If the permission setting is specified as private-include or public-exclude, then a list of users must be specified for inclusion or exclusion respectively. Permission may be granted or denied to an associate member for the read operation only.

Grant

grant entity_name operation (user_name site_name)+

The grant command is used to permit the usage of an operation on an entity by a specified set of users. The users are added to the appropriate inclusion list and if they appear on the corresponding exclusion list they are removed from it. The valid operations to be

granted are assign, checkout, update, derive, name, read_data, set_permission, grant, deny, and global. The general permission value of private, public, private-include, or public-exclude is not altered by the grant command.

Deny

deny entity_name operation (user_name site_name)$^{+}$

The deny command is used to restrict the usage of an operation on an entity by a set of users. The users are added to the appropriate exclusion list and if they appear on the corresponding inclusion list they are removed from it. The operations to be denied are assign, checkout, update, derive, name, read_data, set_permission, grant, deny, and global. The general permission value of private, public, private-include, or public-exclude is not altered by the deny command.