IMPLEMENTATION OF DISTRIBUTED DATABASE AND RELIABLE MULTICAST FOR DISTRIBUTED CONFERENCING SYSTEM VERSION 2

By

AMIT VINAYAK DATE

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001
Copyright 2001

by

Amit Vinayak Date
This work is dedicated to my father Late Vinayak Keshav Date and my grandmother Late Kamlabai Keshav Date
ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my advisor, Dr. Richard Newman, for all his support and guidance throughout my graduate studies. Without his guidance and encouragement this work would not have been possible.

I also thank Dr. Sumi Helal and Dr. Joachim Hammer for their interest on serving on the committee and for their valuable comments. I also want to thank members of DCS team Vijay Manian, Kiran Sirupa, Ashish Bhalani and Ravi for their critique and valuable comments.

Last, but not least, I wish to express my thanks to my mother, Vandana Date, and my brother, Amol Date, for their love and support for my further studies. I dedicate my work to my father, Late Vinayak Keshav Date, and my grandmother, Late Kamlabai Keshav Date. Their faith in me and love will always be a guiding force in my life.
# TABLE OF CONTENTS

| ACKNOWLEDGMENTS | iv |
| ABSTRACT | vii |

## CHAPTERS

### 1 INTRODUCTION
- 1.1 Distributed Conferencing System 1
- 1.2 Motivation 4
  - 1.2.1 Distributed Databases 4
  - 1.2.2 Reliable Multicast 5
- 1.3 Overview 5
  - 1.3.1 Distributed Database 5
  - 1.3.2 Group Communication 5
- 1.4 Organization of the Thesis 6

### 2 STUDY OF RELATED WORK
- 2.1 Distributed Transaction Processing 7
  - 2.1.1 Replication Control Protocols (RCPs) 7
    - 2.1.1.1 Read one write all (ROWA) 8
    - 2.1.1.2 Read one write all available (ROWAA) 8
    - 2.1.1.3 Quorum consensus (QC) 8
  - 2.1.2 Concurrency Control Protocols 9
    - 2.1.2.1 Two-phase locking (2PL) 9
    - 2.1.2.2 Time stamp ordering 10
    - 2.1.2.3 Commit time validation 10
    - 2.1.2.4 DCS approach for concurrency control 11
  - 2.1.3 Replication Control and Concurrency Control Interaction 11
- 2.2 Group Communication 12
  - 2.2.1 FIFO 13
  - 2.2.2 Causal Order Multicast 14
  - 2.2.3 Total-order Multicast 15
  - 2.2.4 Overlapped Groups 16
  - 2.2.5 DCS Approach 17
- 2.3 Conclusion 17
3 DISTRIBUTED DATABASE MODULE

3.1 Introduction
3.2 Requirement Analysis
3.3 DCS Version 2 Approach For Data Consistency and Synchronization
3.4 Design of Database Module
3.5 Implementation Details
3.6 Conclusion

4 COMMUNICATION MODULE

4.1 Introduction
4.2 Requirement Analysis
4.3 Deciding Communication Technology for Interaction between sites in DCS v2
4.4 Design of Communication Module
4.5 Implementation Details for Communication Module
4.6 Interaction with Conference Control Module
4.7 Conclusion

5 TESTING, CONCLUSIONS AND FUTURE WORK

5.1 Testing
5.2 Conclusions
5.3 Future Work

REFERENCES

BIOGRAPHICAL SKETCH
The world is shrinking in size every day. As years go by business decisions are less influenced by geographic location of a vendor. Better means of communication are increasingly becoming the need of the hour. These issues have been the motivation for Distributed Conferencing System. “Distributed Conferencing System” is the brainchild of Dr. Richard Newman who has been working on this concept since 1988 and guided numerous master’s and PhD students in their research endeavors in this exciting arena.

This thesis concentrates on the aspects related to distributed databases and reliable multicast communication.

Distributed database is a union of what appears to be two diametrically opposite approaches to data processing: database system and computer network technologies. The
major objectives behind a database are the desire to integrate operational data of an enterprise and thus provide centralized, thus controlled, access to that data. The technology of the computer networks, on the other hand, promotes a mode of work which goes against all centralization efforts. The key to understanding the symbiosis of these two technologies is to realize that the major objective of database technologies is integrity, and not centralization. A new method has been proposed and implemented to maintain consistency and integrity in our implementation of distributed database for Distributed Conferencing System version 2.

Most high-level network protocols (such as the ISO Transport Protocols or TCP or UDP) provide only a unicast transmission service. That is, nodes of the network only have the ability to send to one other node at a time. All transmission with a unicast service is inherently point-to-point. If a node wants to send the same information to many destinations using a unicast transport service, it must perform a replicated unicast, and send a copy of the data to each destination in turn. To make multicast reliable many protocols have been discussed in the literature. Each protocol offers different degrees of reliability. Causal order multicast protocol has been implemented as per the needs of Distributed Conferencing System version 2.
“Distributed Conferencing System” (DCS) is a distributed system designed to support conferencing over wide area networks (WAN). This system allows geographically separate users to collaborate in preparation of documents, graphics, software tools, as well as demonstrations. Conference control, Database, Communication, Access Control Service, Notification, Decision support modules form the building blocks of DCS. An overview of the functionalities of each of these modules is provided below.

1.1.1 Conference Control subsystem

This subsystem is responsible for “booting up” DCS. It is responsible for initializing other modules, creating conferences, users, merging conferences, deleting conferences, deleting users, providing a graphical user interface (GUI), etc.

1.1.2 Database subsystem

This subsystem provides database services for DCS. In this subsystem a distributed database has been implemented. Integrity and consistency of distributed database are the main considerations for this subsystem.

1.1.3 Communication subsystem

This subsystem provides reliable causal multicast in a conference. All the commands from one host are executed in an order at all sites participating in that conference. It takes into consideration the dynamic needs for the size of the multicast groups, and related issues with loss of messages in the network.
1.1.4 Access Control subsystem

This subsystem deals with access control of issues for the conference. Users in DCS are bound to different roles. Each role has different capabilities. This subsystem maintains an access control matrix to facilitate access decisions.

1.1.5 Notification subsystem

This subsystem is responsible for notifying users of events (e.g. a user logs in, logouts, joins a conference, leaves a conference etc) using means like email, zwrite, mailbox to a user or group of users who are interested in the particular event.

1.1.6 Decision Support subsystem

This subsystem provides templates for making decisions about providing capabilities to a user. Once a decision is made it notifies access control subsystem. Decision making encompasses issues like what should be a quorum, what should be voting methods, how much time should a vote be active, who all should be notified of the decision reached, etc.

These all modules interact and communicate with each other as shown in figure 1 [1]. In this version of DCS all the services have been implemented in machine independent language (JAVA), which will help in integration and portability of these modules on various platforms.
Figure 1.1 DCS System Architecture
In this thesis issues related to distributed database and communication module are dealt in detail.

1.2 Motivation

1.2.1 Distributed Databases

There are several reasons why distributed databases are developed. The following is the list of major motivations [2].

1.2.1.1 Organizational and economic reasons

Many organizations are decentralized, and distributed database approach fits more naturally the structure of the organization.

1.2.1.2 Interconnection of existing database

Distributed database are the natural solution when several database already exist in an organization and the necessity of performing global applications arises.

1.2.1.3 Incremental growth

If an organization grows by adding new, relatively autonomous organizational units, then the distributed database approach supports a smooth incremental growth with minimum degree of impact on already existing units.

1.2.1.4 Reduced communication overhead

In a geographically distributed database the fact that many applications are local clearly reduces communication overhead with respect to centralized databases.

1.2.1.5 Reliability and availability

The distributed database approach especially with redundant data, can be used also in order to obtain higher reliability and availability.
1.2.2 Reliable Multicast

A multicast is a set of nodes that are the common destinations of the same group of messages. The source or sources may be within the multicast group, or may be the other nodes in the network. In DCS a causal order protocol is provided for communication between sites. The motivation behind the communication module is to address the need of reliable communication expected by various modules in DCS.

1.3 Overview

1.3.1 Distributed Database

Two types of databases supported in DCS: local and global. Information that is only relevant to one site is stored in local database while information that is shared among sites is stored in global database. Tables are associated with each conference and they are replicated at all participating sites in a conference. Replicating tables at all sites provides high availability. To increase the availability a strategy of read any and write all available has been used. A scheme is proposed to maintain consistency between the replicated copies in which the site that inserts a record in the table owns the row in the database. All updates to that row are done at this owner site. This helps in avoiding race conditions when multiple sites want to update same set of rows. Freeware Postgres database is used as underling database management system.

1.3.2 Group Communication

This module is responsible for reliable multicast of a message in a particular conference. All messages from one site should be executed in the order issued at all participating sites. This is implemented by maintaining a sequence number at each site and all other sites execute a command only if they have received all previous commands.
from that site. As sites are added and deleted sites from conferences, the multicast group associated with the conference changes. A new multicast group is created with a new version number whenever a site is added or deleted. Communication module primarily supports database module for propagating the commands to various databases to implement the distributed database. Remote Method Invocation (RMI) is used as means of communication between sites after comparing it with socket programming, and CORBA technology provided in java.

1.4 Organization of the Thesis

The next chapter describes the previous work done in this area. The chapter is divided into two sections the first one concerns itself with distributed databases and the second section concerns itself with reliable multicast. Chapter three discusses the design and implementation issues for the database module. Chapter four deals with the design and implementation issues of communication module and the thesis concludes with the chapter five which discusses the conclusions and future work.
2.1 Distributed Transaction Processing

A transaction is a unit of consistent and reliable access to database. It is required that execution of a transaction leads a database from one consistent state to another. A transaction possesses four cardinal properties: atomicity, consistency, isolation and durability, known as the ACID properties [3]. A transaction can be viewed as a series of reads and writes of database items. Since the database is replicated partially or fully, the replication control protocols are developed to govern operations on database items.

2.1.1 Replication Control Protocols (RCPs)

A replication control protocol manages the data object’s distributed components so that its functional behavior is equivalent to that of a single copy. RCP offers the following advantages [4].

- It improves data and system availability, which also improves system fault-tolerance.
- It optimizes performance by accessing local copies instead of remote copies.
- It allows data sharing.

RCP can be divided into two broad categories: pessimistic and optimistic protocols. Pessimistic RCP methods guarantee data consistency during failures by allowing update access on at most one majority partition. Examples are Read One Write All, Read One Write All Available, and Quorum Consensus. Optimistic methods, on the other hand, allow updates on all partitions and use validation upon merge.
2.1.1.1 Read one write all (ROWA)

This approach is value-based, i.e. each site contains a copy of the data object value along with the last operation log. Reads can be done on any copy (preferably the local copy) while writes have to be performed on all copies. This method is attractive in its simplicity. The problem is that it assumes an ideal world and one site failure kills the protocol. Worse, greater the degree of replication, the less availability achieved for the updates. An improvement on this method is Read One Write All Available, which is discussed next.

2.1.1.2 Read one write all available (ROWAA)

This method alleviates the problem of availability for updates in ROWA. For some applications where consistency is not the prime concern but higher availability and efficiency are critical, ROWAA seems to be a perfect solution. In this protocol, data is read from any (preferably local) copy and written data to all available copies. When a partitioned section reconnects to rest of the network, a reconciliation protocol is used to find the latest copy.

2.1.1.3 Quorum consensus (QC)

Like ROWA, QC is also value-based. In QC, copies are assigned non-negative weights \( w[x] \). Database objects are assigned a Read and Write thresholds \( RT[x] \) and \( WT[x] \). Read threshold indicates the number of copies which are required to be read, during a read operation. During a read the copy that has highest timestamp is selected. For a write WT number of copies are written with the write value. Each write quorum of Data object \( x \) has at least one copy in common with every read quorum and every write quorum of \( x \).
Distributed Conferencing System does not require strict adherence of consistency between the replicas of the database. ROWAA was chosen for the implementation as it provides gains with respect to efficiency and ease of implementation.

2.1.2 Concurrency Control Protocols

Concurrency control protocols provide the isolation and consistency properties of transactions. The distributed concurrency control mechanism of a distributed database management system ensures that the consistency of the distributed database is maintained. There are number of concurrency control protocols discussed by M. Tamer Ozsu, Patrick Valduriez [5]. The next section will discuss three of them in particular: two-phase locking, timestamp ordering, and commit time validation.

2.1.2.1 Two-phase locking (2PL)

2PL has the following characteristics.

- It maintains serializability by enforcing mutual exclusions on conflicting operations.
- Database is divided into lockable granules.
- Access to database is interpreted as access to a granule that must be locked first.
- Coarse granularity is pessimistic while fine granularity incurs extra overhead.

In centralized environment, a primary site is dedicated to lock management and all locks requested are directed to that site. When data is replicated, one of those replicas is designated as the primary copy and only that copy needs to be locked for access. In a distributed environment, lock management is decentralized and done by all sites. In case of replicated data, all replicas have to be locked.

As its name implies, 2PL proceeds in two phases. In the acquiring phase, all needed locks are acquired and no locks are released. In the release phase, locks are
released when they are no longer needed and no new locks are acquired. The point that
divides the two phases is called lock point, which is the time when transaction is
committed or aborted. An variation of 2PL is strict 2PL, which requires that all required
locks be released at the lock point.

2.1.2.2 Time stamp ordering

Unlike locking methods, this method does not use mutual exclusion. Instead, a
serialization order is chosen a priori and transactions are executed in that order. This
order is established by a unique timestamp, ts ( T_i ), to each transaction at T_i start up
time. The timestamp are ordered according to the following rule:

Given two conflicting operations O_i, O_k belonging respectively to transactions T_i
and T_k, O_i is executed before O_k if and only if ts( T_i ) < ts(T_k )

The basic timestamp ordering method is a straight implementation of the above
timestamp-ordering rule. If the rule is not fulfilled, one of the conflicting transactions
must be restarted.

One variation on the basic method is called conservative timestamp ordering. In
this method, the operations of transactions are buffered and delayed until the timestamp
ordering scheduler can establish a guaranteed ordering. Restarts are therefore not
possible. However, delaying operations may cause deadlocks, which is certainly not
desirable.

2.1.2.3 Commit time validation

This method is an optimistic approach in oppose to the pessimistic approach the
locking methods adopt. It is optimistic in the sense that it assumes the things are OK most
of the time. In this method, transactions are regarded to consist of three steps: read step,
compute step, and write step. Like in timestamp ordering, timestamps are assigned to the
transactions at start up time. Transactions are then allowed to execute freely reading all items needed, perform computation and decide on the write step. Before installing the updates on the write step, a validation step is performed to check if committing transaction will compromise serializability. Let $T_k$ be a recently committed transaction, this test has the following three cases:

- **Case 1.** All $T_k : ts(T_k) < ts(T_{ij})$ have committed before $T_{ij}$ have started. Validation succeeds in this case.

- **Case 2.** $\exists T_k : ts(T_k) < ts(T_{ij})$ and $T_k$ has completed its write step while $T_{ij}$ is in its write step. Validation succeeds only if $\text{Write}_{-}\text{Set}(T_k) \cap \text{Read}_{-}\text{Set}(T_{ij}) = 0$.

- **Case 3.** $\exists T_k : ts(T_k) < ts(T_{ij})$ and $T_k$ has completed its read step before $T_{ij}$ has completed its read step. Validation succeeds if $\text{Write}_{-}\text{Set}(T_k) \cap \text{Read}_{-}\text{Set}(T_{ij}) = 0$ and $\text{Write}_{-}\text{Set}(T_k) \cap \text{Write}_{-}\text{Set}(T_{ij}) = 0$.

2.1.2.4 DCS approach for concurrency control

In DCS a new strategy has been proposed tailored for the application requirement. Each table and a row is associated with a owner. All updates to the rows take place at the owner sites and are then propagated to all other sites. By this strategy race conditions are avoided that may occur due to simultaneous updates at different sites, as they are serialized at the owner site of the data. Chapter 3 discusses this approach in detail.

2.1.3 Replication Control and Concurrency Control Interaction

The replication control and concurrency control are not two independent mechanisms. Rather they are interrelated and work in concert [4]. Through the replication control, the logical operations on data objects are mapped into sets of physical operations on the object copies while the concurrency control that regulates the access to the copies not the objects. In DCS fully replicated databases have been implemented and ROWAA protocol is adapted for updates. Concurrency is achieved by executing each update at its
owner site. The distributed database uses Communication module, which implements reliable multicast thus assuring each update is received in-order at each site. Thus the database and communication module are tightly coupled with each other.

2.2 Group Communication

The notion of group is essential for development of cooperative software in distributed or autonomous system and has been described in Chow and Johnson [6] and Cordova [7]. The management of a group of process or objects needs an efficient multicast communication mechanism for sending messages to the members of the group. Generically, there are two types of multicast application scenarios. The first is when a client wants to solicit a service from any server who can perform the service. The second is when a client needs to request a service from all members of a group of servers. In the former case, it is not necessary for all servers to respond as long as at least one does. The multicast is performed on a best-effort basis and can be repeated if necessary. The system only needs to guarantee the delivery of the multicast message to the reachable non-faulty process. This is called the best effort multicast. In the latter case, it is often necessary to ensure all the servers have received the request so that consistency of the servers can be maintained. The multicast message should either be received by all of the servers or none of them (i.e., all or none); this is usually called reliable multicast.

Orthogonal to the reliable delivery issue in multicast is the problem of message delivery ordering. When multiple messages are multicast to the same group, they may arrive at different member of groups in different order (due to variable delays in the network). Figure 2.1 shows several group communication examples that require message ordering. G and s represent message and group sources, respectively. Processes may be
outside the group or a member in the group. Ideally, multicast messages must be received and delivered instantaneously in the real-time order that they were sent. Programming groupware would be much simpler if this assumption were true. However, the assumption is unrealistic and meaningless since there is no global time, and the message transfer in the network has a significant and variable delay. The semantics of the multicast can be defined so that the messages received in different orders at different sites can be arranged and delivered to the application process with less restricted rules. The following multicast orderings are listed in increasing order of their strictness:

- **FIFO order**: Multicast messages from a single source are delivered in the order they are sent.
- **Causal order**: Causally related messages from multiple sources are delivered in their order.
- **Total order**: All messages multicast to a group are delivered to all members of the group in the same order. A reliable and total order multicast is called an atomic multicast.

At each site, a communication handler is responsible for message reception and ordered delivery to the application process.

### 2.2.1 FIFO

The FIFO as shown in Figure 2.1(a), is easy to achieve. Because only those messages sent by the same originator need to be ordered, they can be assigned a message sequence number. The message sequence numbers are local to each message source and therefore cannot be used to collate messages coming from different sources, as shown in figure 2(b). Causal and total ordering of multicast messages from multiple sources calls for more sophisticated solutions.
2.2.2 Causal Order Multicast

Two messages are causally related to each other if one message is generated after the receipt of the other. This message order may need to be preserved at all sites since the content of the second message may be affected by the result of processing the first message. This causality may span across several members in a group due to the transitiveness of the causality relationship. To implement the causal ordering of
messages, the sequence can be extended to a vector of sequence numbers, \( S = (S_1, S_2, \ldots, S_n) \), maintained by each member. Each \( S_k \) represents the number of messages so far received from group member \( k \). When member \( I \) multicasts a new message \( m \), it increments \( S_i \) by 1 (indicating the total number of messages that \( I \) has multicast) and attaches the vector \( S \) to \( m \). When receiving a message \( m \) with a sequence vector \( T = (T_1, T_2, \ldots, T_n) \), from member \( I \), member \( j \) accepts or delays the delivery of \( m \) according to the following rules:

- **Accept message** \( m \) if \( T_i = S_i + 1 \) and \( T_k \leq S_k \) for all \( k \neq i \). The first condition indicates that member \( j \) is expecting the next message in sequence from member \( i \). The second condition verifies that member \( j \) has delivered all of the multicast messages that member \( I \) had delivered when it multicast \( m \) (and perhaps several more). So, \( j \) has already delivered all the messages that causally precede \( m \).

- **Delay message** \( m \) if \( T_i > S_i + 1 \) or there exists a \( k \neq i \) such that \( T_k > S_k \). In the former case, some previous multicast message messages from member \( i \) are missing and have not been received by member \( j \). In the latter case, member \( I \) received more multicast messages from some other members of the group when it multicast \( m \) than member \( j \) did. In either case, the message must be delayed to preserve the causality.

- **Reject the message** if \( T_i < S_i \). Duplicated messages from member \( i \) are ignored or rejected by member \( j \). This causal order multicast assumes multicast in a closed group (i.e. the source of multicast is also a member of the group), and multicasts cannot span across groups.

### 2.2.3 Total-order Multicast

Total-order multicast is more expensive to implement. Intuitively, it requires that a multicast must be completed and the multicast messages must be ordered by the multicast completion time before delivery to the application process. Thus it makes sense to combine the atomic and total-order multicast into one protocol. This is the concept behind the two-phase total-order multicast, described as follows. In the first phase of the multicast protocol, the message originator broadcasts messages and collects
acknowledgements with logical timestamps from all the group members. During the
second phase, after all acknowledgements have been collected the originator sends a
commitment message that carries the highest acknowledgement timestamp as the logical
time for the commitment. Members of the group then decide whether a committed
message should be buffered or delivered based on the global logical commitment times of
the multicast messages.

2.2.4 Overlapped Groups

For many distributed applications, a process may belong to more than one group.
Figure 3 shows two equivalent examples of multicast to overlapped groups. The ordering
of messages may be different among disjoint groups even for the same multicast
messages. With overlapping groups, some coordination among groups is necessary to
maintain a consistent ordering of messages for the overlapped members. An example of
where overlapping groups is useful is the implementation of replicated servers
using atomic multicast. One group consists only the servers. For each client, there is
another group, consisting of the client and all of the servers. The clients may belong to
other groups, perhaps obtaining other clients.

![Figure 2.2 Tree representation of overlapped groups](image)
A solution to the problem of overlapping groups is to impose some agreed upon structures for the groups and to multicast messages using the structures. For example, the members of the groups can be structured as a spanning tree (a spanning tree is a suitable representation for group membership in computer network that does not support broadcast in hardware). The root of a tree serves as the leader of the group. The tree edges represent FIFO communication channels. A multicast message is first sent to the leader and then is sent to all members of the group by routing the message through the edges in the tree. Overlapping members must be configured as a common sub tree between two overlapping groups. The example in Figure 3 shows two groups, where group 1 contains members A, B, C, D, and E and group 2 contains members C, D, F and G. The overlap set (C, D) appears as a common sub tree between the two groups.

2.2.5 DCS Approach

In DCS a causal order multicast is chosen, as application requirement do not justify the overhead of total-order multicast. Since multicast groups in DCS are non-intersecting, this indemnifies us from complications of implementation of spanning trees for overlapping groups. Thus conferences in DCS do not share tables. If there are some tables, which are common to two or more conferences, a separate dummy conference for those tables has to be created to maintain reliable multicast.

2.3 Conclusion

This chapter surveys various replication control protocols and concurrency control protocol. This introduction leads us to chapter 3, in which discusses the limitations of these protocols for DCS, and proposes a new strategy for DCS. This chapter also discussed three protocols for group communication: FIFO, causal order and
total order. Chapter 4 discusses the implementation of causal order protocol in which it also addresses the issues of changing membership in a multicast.
CHAPTER 3
DISTRIBUTED DATABASE MODULE

3.1 Introduction

This chapter discusses the requirement analysis, design and implementation details for distributed database. A new approach is proposed for maintaining consistency in distributed database. The services of database module will be used by all modules in DCS.

3.2 Requirement Analysis

Distributed Conferencing system consists of different sites. The main objective of this module is to provide an implementation of a distributed database so as facilitate sharing of data between sites. Each site is assigned a unique site identification number “siteid” by conference control module. Users are also assigned unique user identification numbers by their home site. In the Distributed conference system version 2 there may be users from different sites participating in a conference. The databases at each site will store information pertinent to each conferences with a member whose home is that site. These tables include information needed by notification services, security services, etc. Information relevant only to a particular site is stored locally at each site while information of general relevance is distributed among all the sites. Each conference is also given a unique conference identification number “confid” by conference control module. “Postgresql” a freeware database has been installed at each site. “Postgresql” was chosen, as it is most advanced freeware database available on World Wide Web. All
the tables in DCS are conference-specific and no two conferences can share any tables between them. Thus to share some data at all sites a default global conference is provided that contains all sites in that instance of DCS. Information like available conferences, their IP address, which must be available at all sites, is declared in tables in this global conference. It was decided to fully replicate the databases and defer fragmentation (horizontal / vertical) after usage patterns of the data becomes clearer until the next version of DCS.

3.3 DCS Version 2 Approach For Data Consistency and Synchronization

Replication of databases is desirable in transactional systems like DCS that exhibit high query-to-update ratio and are frequently accessed by several nodes. Replication strategy generally reduces average query response time, at the cost of making updates more complicated by involving all copies of the replicated data. When replicated data stored at different nodes are involved, consistency conflicts among concurrent transactions may not be readily detected. Due to this, it is necessary that all replicated copies must maintain a necessary level of consistency in the presence of update activities.

It is nontrivial to synchronize updates in a distributed environment. Most of the problems are due to the fact that it is costly for any node to evaluate the global state of the transactional system. Several solutions to the fully replicated case have already appeared in the literature (discussed thoroughly in chapter 2 and by Cellary, Gelenbe and Morzy [8]). They had approached the problem through the use of locks and timestamps.

In locking-based approaches, a site acting in behalf of a transaction communicates with all others to inform them about the intended update and to determine whether there are any concurrent updates. This process usually results in driving all sites into
synchronization in order to perform the same update. The disadvantage to this approach is that some updates must be rejected after they have already incurred the expense of inter-site synchronization.

In timestamp-based approaches, the idea is to associate a separate timestamp with each item of the database. The timestamp reflects the time of the most recent update performed on the item. This serves the purpose of ordering the updates applied to a copy of the database in order to preserve consistency. The disadvantage to this approach is the requirement for additional storage necessary for the timestamps themselves, which in some cases, may approximate the size of the database proper itself.

In DCS a new approach to this problem of synchronization is proposed. Transactions in DCS are simple statements like select, create, delete, update which allows concurrency control based on the notion of ownership. All tables are owned by the site at which the command to create the table was issued. A possible approach is to serialize all operations on this table at the site at which it is created. Thus if an update was issued at any other site, it will be first executed at the owners site and then propagated at all sites. Thus race conditions, which may occur due to the simultaneous updates at two sites, can be avoided. The disadvantage of this approach is the one that prevails in all centralized approaches: “single point of failure”. When a site at which the table was created fails all the update activity on the table is suspended till the site recovers. “Centralization” approach also compromises with the advantages of replicated database making availability of the database lower. To increase the availability the granularity of ownership is reduced from table to row level. With this every site, which inserts a row in the database “owns” the row. All updates to that rows can be done at the site where the
row was inserted. With this if a site fails only those rows that were inserted by that site are unavailable, and updates on other rows can proceed. This results in increased availability. To provide row level updates a column called “owner” was added in each table. This column is added to achieve synchronization and hence is made transparent to the users of the database. To achieve the transparency of the owner column, the create command is parsed. A column ‘owner’ is added and a table with ‘_phy’ suffix is created. A view corresponding to table name given by the user in create command is defined which excludes the column ‘owner’.

Commands with global effect like drop table, alter table will still be executed at the site that owns the table and then propagated at all other sites.

3.4 Design of Database Module

DCS consists of globally distributed sites, with each site having its own database. The database module has an interface, which consists of two main methods. One method is for query and other is for update. The query method handles select statement and the update method handles commands like create, drop, insert, delete, update and alter table. The update statement is first executed at table/row owner and then propagated at other sites in a conference. As databases are fully replicated queries, i.e. select statements, are executed locally. The create command creates a table that belongs to a specific conference. Thus syntax to specify conference name is added in standard SQL create table syntax. This is done by adding “@confid” at the end of create table command.

e.g. create table test( emp int, name char, salary int )@10;

Commands like insert, delete, update drop table and alter table are executed first at the site which owns the row/ table and then at all sites in the conference. Thus information
about the table owner and the conference to which the table belongs needs to be
maintained. The ‘glb_conftable_info’ table contains this information. When a drop or
alter table command is received by the database module, first the table owner of the table
is determined. This is done by parsing the command, determining the table name and
finding corresponding owner. If the current site is the table owner, the command is
executed at the current site and then the command is propagated to all other sites in the
conference. An update command can update rows, that belong to different sites. To
ensure that each row is updated at the site that owns it, whenever an update command is
received where clause ‘where owner = siteid’ ia added and the command is propagated
to all sites. Also the original update command is propagated to all other sites in
conference. Other sites just add the where clause ‘where owner = siteid’ to the original
command and propagate the command. Original update command is not propagated in
this case. When a database module receives a propagated command the database module
sets a field while calling the communication module so that the update command is
parsed at all the sites in the conference. The same applies for other commands like delete
that affect more than one row.

The update routine in database interface module calls a parse routine to determine
the nature of the command (insert, update, delete, create, etc.) and to change the
command so that database consistency is maintained according to approach suggested for
DCS databases. The parse routine returns an object ParsedCommand that contains
information specifying to which conference the table belongs, the owner of the table, etc.
The fields in ParsedCommand object and their significance in the design will now be
discussed
Group name/conference name.

This information is used to propagate the command to all the sites in the conference. The parse method parses the command it receives then performs a select on `glb_conftable_info`, which contains with which conference the table is associated. For the create table command the conference name is included in the syntax of the create table command.

Local command (localcmd):

If the owner of the table is not the current site, then no command is executed locally. The command is sent (unicast) to the owner’s site, which processes this command and multicasts it to all sites in the conference, including the current site. For a create command the parse method creates three commands to be executed locally (viz create test_phy table, create view test and add a entry in `glb_conftable_info` table). The field local command is of type String and contains the commands to be executed at local site delimited by ‘#’.

Command to send (cmdtosend):

The field command to send is of type String and contains the command to be executed at all conference site delimited by ‘#’.

Is the current site Owner (wait):

This field is tells if the current site is the owner. If the current site is the owner database module broadcasts the commands to all other sites, if the current site does not own the command it is sent to the owner site.

Is the command required to be parsed global (parseglobal):
Commands that change rows belonging to more than one owner (update, delete) are required to be parsed at each site in the conference. This field is set true for such commands.

The Database Interface module calls the communication module for reliable multicast of the commands to the member sites in DCS.

3.5 Implementation Details

This section discusses how each command is implemented individually in the Database module: It does not describe the syntax of the SQL statements which may be found in Momjian [9] and Ghayal [10].

Create Table

Tables in DCS are conference specific. Thus the suffix “@confid” is added to standard SQL syntax that indicates the conference to which a table belongs. A typical example of a create statement is:

```sql
create table test( emp int , name char, salary  int )@10;
```

Here `test` is table name, `emp`, `name` and `salary` are attribute names. `10` is the conference id in which this table will be created. The database command produces following commands after parsing the create command.

```sql
create test_phy( emp int , name char, salary int, owner int ) @10
create view test AS select emp ,name, salary from test;
insert into glb_conftable_info values(conferenceid, tablename ,siteid )
```

These commands are then propagated to all member sites in the conference. The following fields are set in the `ParsedCommand` object passed to the database module by the parse program.
groupname = 10. Table test belongs to conference whose confid is ‘10’.

cmdtosend = create test_phy( emp int , name char, salary int, owner int) @10
create view test AS select emp ,name, salary from test
insert into glb_conftable_info
values(conferenceid, tablename ,siteid )

localcmd = create test_phy( emp int , name char, salary int, owner int) @10
create view test AS select emp ,name, salary from test
insert into glb_conftable_info
values(conferenceid, tablename ,siteid )

wait/owner = true ( current site is owner of the table)
parsesglobal = false ( commands are not required to be parsed at each member site)

The DB subsystem creates a view that contains all the fields that were given by the user in the create command. Thus the user is provided with abstract view of original table.
The user of the distributed database module is not aware of the table with the “_phy” suffix and issues all commands on the view.

Select

The select command will be called from the executeQuery method of the
DatabaseInterface module. The select command by the user will always be on the view.
As databases are fully replicated all the select queries are always executed locally.

e.g. select * from test;

Insert

The command issued by the user on insert will be:

    insert into test values ( 99, 'Amit', 100);

where 99 is emp#, Amit is his name and his salary is 100. As the user is not aware of a
table test_phy the insert command is issued on the view test. The insert command will be
parsed by the \textit{parse} method and the table name \textit{“test”} will be determined. The insert command will be changed to:

\begin{verbatim}
insert into test_phy values ( 99, 'Amit', 100, 12);
\end{verbatim}

Where ‘12’ is \textit{siteid} inserted by the parse module. The value of owner field is added and the insert is done on the corresponding physical table. The changes are automatically reflected in the view.

The value of the \textit{ParsedCommand} object returned are:

\begin{verbatim}
grpname = 10  ( The table test belongs to conference whose confid is 10. This information will be used by communication module to multicast the command to other sites in conference)

cmdtosend = insert into test_phy values ( 99, 'Amit', 100, 12);
The command which inserts into the physical table test_phy is propagated)

localcmd = insert into test_phy values ( 99, 'Amit', 100, 12);

wait/owner = true  ( current site is owner of the row which is being inserted)

parseglobal = false ( commands are not required to be parsed at each member site)
\end{verbatim}

\textbf{Update Rows}

The command issued by the user will be update test set salary =200 where emp# =99. Thus with this command user wants to change salary of all employees whose emp# is 99 to 200. The values of the \textit{ParsedCommand} object will be:

\begin{verbatim}
grpname = 10. Table test belongs to conference whose confid is ‘10’
cmdtosend = update test set salary =200 where( emp# =99 and owner = 12 ) # update
test set salary =200 where emp# =99
\end{verbatim}
The update command can change rows that are owned by more than one site. Thus database module adds the clause ‘where owner = siteid’ at the end of the command. Both the original command and the command with the where clause are propagated to all sites. wait/owner field is not significant for delete because a update command can affect rows with more than one owner

parseglobal = true

A site that receives a command with parseglobal set to true, it initializes databaseinterface in a special mode. In this mode the database interface at that site creates a update command

\[ \text{update test set salary =200 where( emp# =99 and owner = 14 )} \]

The siteid of current site is 14 and this command is propagated to all sites.

Delete Rows

The delete command is very similar to the update command. The command issued by the user will be:

\[ \text{delete from test where emp# =99.} \]

Thus with this command, the user wants to delete all employees whose emp# is ‘99’. The values of ParsedCommand object will be:

\[ \text{groupname = 10. Table test belongs to conference whose confid is ‘10’}. \]

\[ \text{cmdtosend = delete from test Phy where( emp# =99 and owner = 12) # delete from test where emp# =99} \]

The delete command can delete rows that are owned by more than one site. Thus database module adds the clause ‘where owner = siteid’ at the end of the delete
command. Both the original command and the command with \textit{where} clause are propagated to all sites.

\texttt{wait/owner} field is not significant for delete command because a update command can affect rows with more than one owner.

\textit{parseglobal} = \textit{true}

A site that receives a command with \textit{parseglobal} set to true, it initializes \textit{databaseinterface} in a special mode. In this mode the database interface at that site creates a delete command

\texttt{delete from test where( emp\# =99 and owner = 14 )}

The siteid of current site is 14 and this command is propagated to all sites.

\textbf{Drop Table}

The command issued by the user will be:

\texttt{drop table test.}

The parse method will parse this command and determine the table name, corresponding conference name and the table owner. The \textit{parsedCommand} object will have following values:

\textit{groupname} = 10. \textit{Table test belongs to conference whose confid is ‘10’}.

\textit{Cmdtosend}: The value of this field will depend upon whether the current site is owner of the table. If the current site is the owner it will have the following value:

\texttt{drop table test#drop table test\_phy# delete from glb\_conftable\_info where tablename = test.}

The table and the view are deleted, also the entry for this table in \textit{glb\_conftable\_info} is deleted. If the current site is the owner the original command:
*drop table test*

is unicast to the owner site.

*Cmdlocal*: The values of this field will depend upon whether the current site is owner of the table. If the current site is the owner it will have following value:

```
drop table test#drop table test_phy# delete from glb_conftable_info where tablename = test.
```

If current site is not the owner no command is executed locally. The remote owner site after executing will broadcast the command to all the member site including the current site.

*Owner/wait*: ‘true’ if the current site is owner. If some remote site is the owner of the table the command is unicast to that site.

*Parseglobal*: The value of this field is ‘false’ as this command does not need to be parsed at all sites.

*Alter Table:*

The commands issued by the user are:

```
alter table test rename test to testrenamed
alter table test add newcol int;
```

In the current version of *postgres*, the drop column command is not supported. The parse method will parse the *alter* command and the returned object *ParsedCommand* will have following values:

*groupname* = 10. *Table test belongs to conference whose confid is ‘10’.*

*Cmdtosend*: The values of this field will depend upon whether the current site is owner of the table. If the current site is the owner it will have following value
alter table test rename test to testrenamed#
alter table test_phy rename test to
testrenamed_phy#
update glb_conftable_info set tablename testrenamed where tablename
= test.

The view and the table names are changed, also glb_conftable_info table is changed to
reflect the new table name. If the alter command adds a column in the schema database
module changes both the view and the physical table. In this case the contents of
glb_conftable_info don’t need to be changed. If the current site is not the table owner the
original command is uni-cast to the remote site owner.

Cmdlocal: The cmdlocal will have same values as cmdtosend if the current site is table
owner, else no command is executed locally. The remote site will execute the command
at its site and propagate it to all sites in the conference including current site.

Owner/wait: ‘true’ if the current site is owner. If some remote site is the owner of the
table the command is unicast to that site.

Parseglobal: The value of this field is ‘false’ as this command does not need to be parsed
at all sites.

3.6 Conclusion

This chapter describes the strategy used in DCS for database consistency in
replicated databases. Each row in the table will be owned by a site. All the updates to
tables are executed at the table owner/row owner site. To implement this strategy an
extra column ‘owner’ was introduced. This column is made transparent to end user by
creating a view corresponding to table name given in create command. The commands
which affect more than one rows (update, delete) are very costly, for each command
database module generates \( n(n+1) \) commands. \( n \) are the number of sites in the
conference). Select are most efficient and are executed locally as databases are fully replicated. The cost of insert, create, drop commands is $n$. The database module is closely coupled with the communication module. The next chapter describes the design and implementation issues of communication module.
CHAPTER 4
COMMUNICATION MODULE

4.1 Introduction

This chapter discusses the requirement analysis, design and implementation details for communication module. This module implements a causal order, reliable multicast. This module will be primarily used by Database to multicast the database commands to sites in the conference.

4.2 Requirement Analysis

The main objective of the communication module is to ensure that each member in the conference receives all database commands issued at any site. Each conference in DCS can be made up of one or more sites. Each conference has its own set of tables. Thus a conference specific multicast is required. Also sites can join or leave conferences at random. Thus the multicast should dynamically adapt to varying size of each conference.

4.3 Deciding Communication Technology for Interaction between sites in DCS v2

Communication between sites in Distributed Conferencing System version 2 can be done using technologies like Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Remote Method Invocation, and Common Object Request Broker Architecture (CORBA). This section summarizes the similarities and differences between them, which helps in understanding these technologies and make choosing between them easier.
TCP and UDP

TCP and UDP are the transport layer protocols in the Internet Protocol stack [11]. The primary difference between UDP and TCP is that UDP does not necessarily provide reliable data transmission. In fact, there’s no guarantee by the protocol that the data will even arrive at its destination. UDP is effective and useful in many ways when the goal of a program is to transmit as much information as quickly as possible, where any given piece of the data is relatively unimportant.

The purpose of TCP is to provide data transmission that can be considered reliable and to maintain a virtual connection between devices or services that are “speaking” to each other [12]. Lower network layers treat every packet like a separate unit; therefore, it’s possible for packets to be sent along completely different routes, even though they’re all part of the same message. TCP is responsible for data recovery in the event that packets are received out of sequence, lost, or otherwise corrupted during delivery. It accomplishes this recovery by providing a sequence number with each packet that it sends.

RMI & CORBA IDL v/s UDP &TCP-IP

Using RMI, Java objects can invoke the methods of remote objects running under an entirely different JVM, as if they were locally available. RMI is inherently a socket solution and is built on top of lower level transport layers. In general RMI can't be any faster than sockets. Transferring the same object using sockets is at least two times faster than using RMI. Performance of RMI is strongly dependent upon the implementation of JVM, the class library and the platform.
RMI is built on top of Object Serialization (OS). OS is simply one way of passing data around, and, like RMI, it too is quite general: One can pass any suitable prepared object over a network connection and it will show up on the other side, intact and ready to have its methods called. The same arguments about generality and efficiency that applied to RMI apply here as well.

On the other hand, Object Serialization does supply some optimizations of its own. For example in certain circumstances, when user code sends an object twice, the underlying OS layer will send the full object the first time and will only send an abbreviation the second time: "Hey, object #45345 sent again." This technique can save bandwidth, but one can use the same technique in code as well.

In conclusion, there are two issues involved: efficiency and ease of programming. RMI can be very convenient if your protocol resembles function calls; on the other hand, being very general, RMI will probably have poorer performance, as compared to a finely-tuned custom solution.

RMI and CORBA [13]

Because RMI and CORBA have similar purposes, RMI and Java IDL (CORBA) have some similar features and capabilities—as well as some differences [13].

100% Pure Java vs. Support for Legacy Applications

Java RMI is a 100% Pure Java solution for remote objects, providing all the advantages of Java’s "write once, run anywhere" abilities. Servers and clients developed with Java RMI can be deployed anywhere on a network on any platform that supports the Java runtime environment.
Java IDL (CORBA), in contrast, is based on an industry standard for remotely invoking objects written in any supported programming language. As a result, Java IDL provides a way to connect to "legacy" applications that still serve vital business needs but that were written in languages other than Java.

Communication Protocols [13]

Java RMI and Java IDL (CORBA) currently use different protocols for communicating between objects on different platforms. Java IDL uses the CORBA-standard Internet Inter-Orb Protocol (IIOP), the protocol shared by all CORBA-compliant Object Request Brokers. Together with IDL, IIOP enables objects residing on diverse platforms and written in diverse languages to interact in standard ways. Java RMI currently uses the Java Remote Messaging Protocol (JRMP)--a protocol developed specifically for Java's remote object capabilities.

For the future, Sun and IBM have announced plans to enable RMI to use the IIOP protocol to communicate with CORBA-compliant remote objects.

Objects by Reference, Objects by Value [9]

In Java IDL (CORBA), a client interacts with a remote object by reference. That is, the client never gets an actual copy of the server object in its own runtime environment. Instead, the client uses stubs in the local runtime to manipulate the server object residing on the remote platform.

In contrast, RMI enables a client to interact with a remote object by reference, or to download it and manipulate it in the local runtime environment by value. This is because all objects in RMI are Java objects. RMI uses the object serialization capabilities of the Java language to transport objects from the server to the client. Java IDL, because
it interacts with objects written in any language, can't take advantage of this "write once, run anywhere" feature of the Java programming language.

Future versions of the CORBA specification will include protocols for passing objects by value.

Before making decision which of these methods to following questions were answered.

- How critical is efficiency? Is tradeoff b/w : efficiency and ease of programming acceptable?
- Serialization provided in java can be a benefit?
- How critical is portability?
- Will applications will be invoked remotely?
- Which method addresses the security issues as requires by DCS ver2?

In DCS application, where efficiency is not very crucial ease of programming is more desirable goal. Most of the remote calls will be function calls and java serialization can be a huge benefit. As DCS is coded in java, and all applications are also coded in java complications arising due to cross platform applications need not be considered. Conference control may have to invoke remote applications. With these considerations, it was decided to use RMI for communication between sites. DCS ver2 will have a security module, which will address security issues for RMI communication.

4.4 Design of Communication Module

The communication module will be used for communication between sites by the Database module. For each conference in DCS, has associated databases. Hence there will be a multicast group associated with each conference. For each multicast group
causal order protocol is implemented. To achieve this a sequence number is associated with each message. Also each site maintains a vector in which it maintains the highest in order sequence number from all sites in the conference. Thus a site executes a message from a site when it has received all the messages (from all sites) which that the other site has received or sent before the current message [6].

Tables associated with each conference are unique, i.e. no two conferences share same tables. As each conference has its own multicast group, this assumption is necessary to maintain the causal order. As messages are to be multicast to all sites in the conference information about the IP address and port number for each site is required. The information is stored in the table glb_site_info and has attributes site identification number (siteid), site IP address (site-ip), and port number (portno). The membership in a conference is dynamic. Thus communication module maintains a table containing which sites are present in a conference. Whenever a site joins or leaves a conference a different multicast group is created. This information is stored in glb_conf_info, which has attributes conference identification number (confid), site identification number (siteid) and version number (verno).

When a site in a conference has to broadcast a message to sites within a conference, communication module finds all the sites with a given conference name and max version number and broadcasts message as a sequence of uni-cast messages to each site. When new version of multicast is being created, there is a time in which only few entries of new multicast group have been updated and the database is inconsistent stage. If during this time a message is broadcast, it would not be sent to all the sites in the multicast as the update of new version of multicast was incomplete. To address this
problem of inconsistent database stage during a update, a row with siteid equal to −1 is introduced which signifies the end of the update. The maximum current version of a conference multicast is the maximum version number for the siteid “-1”. So only completed updates will be reflected and no messages will be lost during addition and deletion of sites in the conference.

When the conference is initialized a initial version number is assigned to it. Thus at t=0 the contents of glb_site_info are shown in figure 4.1 and the contents of glb_conf_info are shown in figure 4.2.

<table>
<thead>
<tr>
<th>siteid</th>
<th>Ipaddr</th>
<th>Port no</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>128.227.176.71</td>
<td>7000</td>
</tr>
</tbody>
</table>

Figure 4.1 glb_site_info at t=0

<table>
<thead>
<tr>
<th>Confid</th>
<th>siteid</th>
<th>Version no</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.2 glb_conf_info at t=0

When at t =1 user at site b decides to join the conference the contents of glb_site_info are shown in figure 4.3 and the contents of glb_conf_info are shown in figure 4.4.
When at $t=2$ all users at site b leave the conference the contents of $glb\_site\_info$ are shown in figure 4.5 and the contents of $glb\_conf\_info$ are shown in figure 4.6.
Once a site receives a message it checks whether it has already received all messages that site has received, and the current message number is one more than the last message it has received from the site sending message. The site also ensure that the two messages have same version number.

Consider a site ‘0’ with it at time t having vector myarray for particular conf# and version# (figure 4.7)

<table>
<thead>
<tr>
<th>#0</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Site 0 receives a message from site 3 for this conf# whose msgarray is as shown in figure 4.8.
From the figure 4.8 site 0 can infer that site 3 has received more messages from site # 2, than it has received. Site 0 pulls this message from site 3. Site 0 uses site name and message number as message identification parameters as site number of a site changes from version to version. Also it insures that it has received all other messages from sent and received by site 3. Three approaches can be used to retrieve the messages: pull, push, pull & push. In the pull approach (implemented in DCS ver 2) for its simplicity ) a site retrieves lost messages only if it receives a message from a site who has received more messages than what it has received. In push approach a site sends messages to site if other site hasn’t received any message it has received. The “push/pull” is the safest approach, which is combination of above two approaches.

4.5 Implementation Details for Communication Module

The format of the message in DCS ver2 for communication from one site to another is:

Message:

`confno` (type: integer): The conference number for which the message is sent.

`versionno` (type: integer): The version number for current multicast group.

`msgarray` (type: vector): This contains information of which messages this site has received from all other sites.

`msgsiteno` (type: integer): The site number of this site in the `msgarray`. 
msgsitename (type: integer): Unique siteid assigned by conference control to this site.

cmd (type: String): This is the database command to be executed.

parseglobal (type: boolean): This field indicates whether the database command needs to parsed at each site.

type (type: String): This is field has value “update” as databases are fully replicated and elect”) will be executed locally. The field is designed for the future where databases might not be fully replicated.

database (type: String) : As of now the there are two databases at each site, local and global.

The two most important data structures at each site are myarray and cmdarray. myarray is a four-dimensional data structure. The first dimension is for the number of conferences in the site, the second dimension represents the version numbers for each conference, and the third and fourth dimension are an array representing what messages this site has received from every other site in the conference. This data structure is implemented as vector containing vectors in java.

For conference 0 and version ‘n’ the array for site0 is shown in figure 4.9.

<table>
<thead>
<tr>
<th>#0</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>#1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>#2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>#3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.9 myarray
cmdarray is a four-dimensional data structure. The first dimension is for the number of conferences in the site the second dimension represents the version numbers for each conference, and the third and fourth dimension are an array containing messages received from other sites. This data structure is implemented as vector containing vectors in java.

For conference 0 and version ‘n’ the array for site0 is shown in figure 4.10.

<table>
<thead>
<tr>
<th>#</th>
<th>Msg0</th>
<th>Msg1</th>
<th>Msg2</th>
<th>--</th>
<th>Site 0 has received this messages from site 0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Msg0</td>
<td>Msg1</td>
<td>--</td>
<td>--</td>
<td>site 0 has received this messages from site 1</td>
</tr>
<tr>
<td>#2</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>site 0 has received this messages from site 2</td>
</tr>
<tr>
<td>#3</td>
<td>Msg0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>site 0 has received this messages from site 3</td>
</tr>
</tbody>
</table>

Figure 4.10 cmdarray

Following section discusses the main routines in the communication module:

Add Site

This remote routine will be called by conference control to add new site in a conference.

The signature of the routine is:

```java
public boolean addsite(int confid, int stname, String ipaddr, int portno ) throws java.rmi.RemoteException;
```

To add a new site first the highest version number for the current conference is determined. If this is the first site in the conference an entry added for this site and a default entry with siteid ‘-1’ in the glb_conf-info table. If the conference already exists the version number of the conference is incremented by 1 and the new site is added to the existing sites. The new list is written to the database. The glb_site-info table is also updated with siteid, ipaddr and portno for this new site. Send group message is then
invoked with command type as new version. Thus the new site is added at all sites in the conference.

**Delete Site**

This remote routine will be called by conference control to add new site in a conference. The signature of the routine is:

```java
class DeleteSite {
    public boolean deletesite(int confid, int stname) throws java.rmi.RemoteException;
}
```

To delete a site is added the method first finds the highest version number for the current conference. The version number of the conference is incremented by 1 and the new site list excluding the deleted site is multicast to all members in the site. The information of this site is deleted from `glob_site_info` table. Send group message is then invoked with command type as new version. Thus the site is deleted at all sites in the conference.

**Send Group Message**

This remote method is called by database module, add site and delete site methods to propagate the database command to all sites in the conference. The signature of this method is:

```java
class SendGroupMessage {
    public boolean sendgrpmsg(String cmd, String type, int parseglobal, int toconfno)
            throws java.rmi.RemoteException;
}
```

`sendgrpmsg` creates a message to be passed to all the sites. It finds the maximum version number for current conference. It increments the number of messages received by the current site by one for this conference and version number. The multicast is implemented as N-1 uni-casts where N is the number of sites in the conference. For each unicast `sendgrpmsg` finds the member site’s IP address and port number from the `glob_site_info` table. The `siteid` for the sites in this conference are queried from the `glob_conf_info` table.
It also forms the message to be sent to the remote site, which contains the database command to be executed. *sendgrpmsg* calls the receive group message routines at the remote sites.

**Receive Group Message**

This remote method is called by *sendgrpmsg*. The signature of the method is:

```
public void receivegrpmsg(Message msg) throws java.rmi.RemoteException
```

This method receives the command from remote site. It then verifies receiving site has received all the messages that the sending site has received, and that the current message is the next in sequence number from that site (i.e. it checks that current message is in causal order). If not it pulls the messages it has not received by remotely calling *givemsg*.

Once it has ensured the message is in causal order, it calls the *databaseinterface* routine and parses the command if “parse global” field is set to true in the message, else it executes the message locally.

**Retrieve Specified Message**

This remote method will be called from *receivegrpmsg*. The signature of the method is

```
public Message givemsg(int msgnorow, int msgnocol, int confid, int verno) throws java.rmi.RemoteException;
```

This method returns the message specified by conference identification number, version number and row and column number signifying the column\textsuperscript{th} message from row\textsuperscript{th} site.

**Send Unicast Message**

This routine will be called from the database module to send the command to be executed at owners site. The signature of the method is:

```
public boolean sendmsg(String cmd, int tositename) throws java.rmi.RemoteException;
```
This method finds the IP address and port number of the owner site from `glb_site_info` table. It calls `receivemsg` at the owner’s site.

### Receive Message

This method will be remotely called by send uni-cast message. The signature of the method is

```
public boolean receivemsg(Message msg) throws java.rmi.RemoteException;
```

This method calls `databaseinterface` module. This is site is owner site of the command. The database module will in turn call `sendgrpmsg` and send the command to all sites in the conference.

### Garbage Collection

This method will be periodically called by conference control module. This method cleans up all the messages that are known to have been received by all sites. This is done by finding out the minimum number from each row and purging all the commands till that number. For e.g. for the `myarray` shown in the figure 4.9, inspecting column one it can be inferred that all sites have received two messages hence `msg0` and `msg1` can safely be purged from `cmdarray` (figure 4.10).

### 4.6 Interaction with Conference Control Module

A user using the GUI provided by conference control services creates new conferences in DCS. The conference control module has to provide a unique conference identification number for each conference. There should be a mapping between the conference name and conference identification number. Whenever a new site is initialized, a communication module must be started. Whenever a new site is added in a
conference or a site is deleted, the multicast group of the conference should be updated by conference control.

4.7 Conclusion

This chapter describes the design and implementation of communication module in Distributed Conferencing Control version 2. This module runs as a RMI service and provides reliable conference specific multicast. It implements the causal-order multicast protocol to provide reliability so that causally related messages from multiple sources are delivered in their order. The causal order protocol is implemented by maintaining a vector for each conference that indicates which messages have been received by that site. Each site also maintains a vector that stores all the messages received at that site. Whenever a new site is added a new list of sites with the added site is created, the new list is given a new version number and multicast to all sites in the conference. For a multicast the site sending message determines the site-list with highest version number and multicast is implemented as multiple uni-cast’s. Similarly when a site is deleted a new-list with new version number excluding the deleted site is multicast to all sites in the conference.
CHAPTER 5
TESTING, CONCLUSIONS AND FUTURE WORK

5.1 Testing

Sound software engineering policies were used for the development of database and communication modules [14]. The design of these modules began with group discussions in weekly DCS meetings. Requirement analysis was done by studying the needs of other modules and necessary features were incorporated in the database module and communication module. Initial design documents for the distributed database and the communication module were thoroughly scrutinized in DCS meetings. The use of these software engineering practices resulted in a well-documented code that met its requirements. Unit test programs were designed to test each functionality. The testing was done in Network Security Lab in University of Florida. Machines named “ripley” and “jekyll” were used to test the modules and ‘postgres’ was installed on this machines.

Add Site

In this test program sites are added to a conference. If the added site is the first site in the conference a new array is initialized for this conference. A conference was started at ripleys and jekyll was added to the conference.

Delete Site

This test program deleted a site from the conference. Site ‘jekyll’ was deleted from the conference during the testing.
Create Table

A table was created using the interface provided by database module at ripley. As jekyll was part of the conference the table was also created successfully at that site. Both the table with suffix ‘_phy’ and the view were created also glb_conftable_info was updated with table name, table owner and conference name. The test which was executed at ripley created a table ‘test’.

create table test (a int2, b int2)@10

Tables test_phy and view test were created at both sites. The ripley was assigned a siteid 123 by addsite routine. The entry in glb_conftable_info was table name (test), conference identification number (10) and table owner (123).

Insert Row

A row was inserted at each site ripley and jekyll. Appropriate siteid was inserted in the owner column of the table with ‘_phy’ suffix. Command issued during testing was

insert into test values ( 2,2)

This command was executed at both ripley (siteid 123) and jekyll (siteid 11). The contents of test_phy are shown in figure 5.1.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>123</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 5.1 Table test_phy after insert
Update Rows

This command updates the rows in the table. Rows owned by both sites were changed. As table with suffix ‘_phy’ was changed view was automatically updated. The command issued at *ripley* was:

```
Update test set b = 6 where b = 2;
```

The contents of *test_phy* after execution of the command are shown in figure 5.2.

![Table test_phy after update](image)

Delete Rows

This command deletes the rows from the table. Rows owned by both sites were deleted. As table with suffix ‘_phy’ was changed view was automatically updated. The command issued at *ripley* was:

```
delete test where b = 2;
```

The contents of *test_phy* after execution of the command are shown in figure 5.3.

![Table test_phy after delete](image)
Alter Table

This command was used to change the table name. The table and the corresponding view names were changed at both the sites. Also the *glb_conftable_info* was updated with new table name. The command issued at *ripley* was:

\[
\text{Alter table test rename to test1;}
\]

Tables *test1_phy* and view *test1* was created. After the command the entries in *glb_conftable_info* were table name (*test1*), table owner (123), conference name (10).

Drop Table

This command was used to drop the table. Both the table with suffix '_phy' and view were dropped. The command issued at *ripley* was:

\[
\text{Drop table test;}
\]

Tables *test_phy* and view *test* were dropped also entry corresponding to table test was deleted from *glb_conftable_info*.

Also these modules are used by all other modules in DCS they were extensively tested by the developers of conference control, access control notification services sub systems.

5.2 Conclusions

The objective of this thesis was to provide a distributed database implementation and reliable communication for various modules in distributed conferencing system. Both the modules have been successfully implemented. These modules are the backbone of DCS with almost every other module using their services.
5.3 Future Work

Horizontal and Vertical Fragmentation of Database

In this version of DCS fully replicated databases have been implemented. Space requirements can be reduced by using techniques of horizontal and vertical fragmentation.

Protocol for Change of Ownership in Database

Study has to be done on various issues arising when a site owning some rows leaves the conference. A protocol for smooth transfer of ownership has to be devised and implemented.

Count to Infinity

In communication module integers are used for counting the number of messages received by each site. As integers in java can count finite numbers a mechanism has to developed and implemented to set the count to zero when it reaches the maximum number represented in java.

Message Consistency

A total order multicast can be implemented to provide stricter consistency for message communication between sites.

Inter-site Communication Security

Security issues for RMI need to be studied and secure RMI must be developed for interaction between sites.

Inter-group Multicast Communication

In current implementation a table belongs to a single conference as overlapped multicast groups are not supported. Future implementations should remove these limitations.
The database and communication module are now functional in the version 2 of DCS. Conference Control module, Access Control Module, Notification module are being developed by the team members of DCS and will be integrated with database and communication module soon. The tasks suggested in future work can then be implemented as per the priority of needs of DCS users.
REFERENCES


BIOGRAPHICAL SKETCH

Amit Vinayak Date was born in Pune, India. He received his undergraduate degree in computer engineering from Ramrao Adik Institute of Technology, Bombay University, India. Upon graduation, he joined Larsen and Toubro Information Technology Ltd as a software engineer. He worked at New Brunswick Telephone Company, Canada, as a software consultant.

In December 1999, he left his job to pursue a Master of Science in computer and information science, at the University of Florida.